From definition to application —— A simple introduction to Markov Chain Monte Carlo

Jiayu He, Yuyin Li, Qingli Zeng, Yaokun Li

***Abstract*—This article discusses the definition of Markov Chain Monte Carlo Algorithm, basic usage on Metropolis-Hasting and Gibbs Sampling, and introduces two papers of MCMC application about solving matrix computation and finding the minimum node separator.**

***Index Terms*—Algorithms, MCMC**

I. Introduction

CONP, ...

II. Definition

A. aaa

B. bbb

```
import random
```

III. Metropolis-Hasting

A. aaa

B. bbb

IV. Gibbs Sampling

A. aaa

B. bbb

V. Application1: Enhancing Monte Carlo Preconditioning Methods for Matrix Computations

A. Introduction

This paper [1] introduces a MC random algorithm for preconditioning to find an inverse of matrices or to solve the following linear equation:

$$Ax = b$$

where A is a matrix with m rows and m columns and b is a matrix with m rows and 1 column. x is a $m * 1$ matrix $[x_1; x_2; \ldots; x_m]$. We need to calculate x with the given A and b.

It's called the problem of solving systems of linear algebraic equations (SLAE). The direct method is slow for large-scale equations, so we introduce a Monte Carlo method to quickly yield a rough estimate of the preconditioner, as an alternative to the MSPAI algorithm.

B. Direct method

(1) Gaussian elimination

A naive method is Gaussian elimination, which is usually taught in the first class of linear algebra.

We perform a sequence of elementary row operations to modify the matrix until the lower left-hand corner of the matrix is filled with zeros.Then, using back-substitution, each unknown can be solved for.

The most time consuming part is multiplication, there are totally $\frac{2n^3+3n^2-5n}{6}$ multiplications, so the complexity is $O(n^3)$. It is not fast enough for solving practical problems.

(2) Iterative method

A more efficient way to solve the problem is through iterations. Iterative solvers are usually used to compute the solutions of these systems due to their predictability and reliability since they can never go wrong. However, for large-scale problems, they can be very time consuming.

A typical iterative algorithm is Jacobi method. We decompose A in a diagonal component D, and a remainder R in the form of $A = D + R$. D only has non-zero values in the principal diagonal and R only has zero values in it.

We obtain the solution iteratively via the following equation:

$$x^{k+1} = D^{-1}(b - Rx^{(k)})$$

where $x^{(k)}$ is the kth approximation of iteration and x^{k+1} is the next iteration of x.

We input a $x^{(0)}$, the preconditioner, as a guess to the solution, and iterate x until the convergence is reached. We use the following equation for calculation in practice:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)})$$

The most time consuming part is still multiplication, there are $7n^2 + 7n$ operations in the main loop[2], the complexity is $O(kn^2)$, where k represents the number of iterations better than Gaussian elimination.

An alternative iterative method is Gauss-Seidel method, it converges faster than Jacobi method by using the most recently available approximations of the elements, the complexity is still $O(kn^2)$. To improve their performance, we can figure that the iteration times can be reduced, so it's important to find a preconditioner which is closer as possible to the solution. Therefore, these algorithms often rely on preconditioners to speed up the computations and to ensure

faster convergence.

C. SPAI algorithm

The SParse Approximate Inverse Preconditioner (SPAI) is used to compute a sparse approximate inverse matrix M for a given sparse input matrix B.The algorithm explicitly computes the approximate inverse, which is intended to be applied as a preconditioner of an iterative method. Since there will be an input $x^{(0)}$ for the iteration, if it differs a lot with the solution of x, obviously we can not reach convergence in a short time. SPAI helps us to determine a better preconditioner.
The original SPAI algorithm[3] computes the preconditioner M explicitly by minimizing $\|AM - I\|$ in the Frobenius norm. Since

$$\|AM - I\|_F^2 = \sum_{k=1}^{n} \|Am_k - e_k\|_2^2$$

The solution of it separates into the n independent least-squares problems for the sparse vector m_k:

$$\min_{m_k} \|Am_k - e_k\|_2, \quad k = 1, \ldots, n$$

The SPAI algorithm begins with a diagonal pattern. It then augments progressively the sparsity pattern of M to further reduce each residual $r_k = e_k - Am_k$. Once the new entries have been selected and added to m_k, the least-squares problem is solved again with the augmented set of indices. The algorithm proceeds until each column m_k of M satisfies

$$\|Am_k - e_k\|_2 < \varepsilon$$

where ϵ is a tolerance set by the user; it controls the fill-in and the quality of the preconditioner.
Since SPAI is introduced in 1996, several advances building upon the initial implementation have been made. Modified SParse Approximate Inverse Preconditioner (MSPAI) is used as a preconditioner for large sparse and ill-conditioned systems of linear equations. Factorized SParse Approximate Inverse (FSPAI) is highly scalable and it's applicable only to symmetric positive definite systems of this kind.
In this paper, the author compare our algorithm with the MSPAI algorithm. The MSPAI extended the basic SPAI minimization

$$\min_M \|AM - I\|_F^2$$

to target form and further generalized it in order to add additional probing constraints:

$$\min_M \left\| \begin{pmatrix} C_0 \\ \rho e^T C_0 \end{pmatrix} M - \begin{pmatrix} B_0 \\ \rho e^T B_0 \end{pmatrix} \right\|_F^2$$

D. Monte Carlo Approach

As mentioned in class, Monte Carlo methods are probabilistic methods, that use random numbers to estimate the solution of a problem. They are good candidates for parallelisation because of the fact that many independent samples are used to estimate the solution. These samples can be calculated in parallel, thereby speeding up the solution finding process.

(1) Main properties
The author think the MC approach must have these properties while designing the algorithm to show a better performance than the traditional methods with parallel computation.
1. efficient distribution of the compute data.
2. minimum communication during the computation.
3. increased precision achieved by adding extra refinement computations.

(2) Algorithm for calculating inverse
The following procedure allows to extend the Monte Carlo algorithm for processing diagonally dominant matrices to get the inverse.
Let B be a sparse, square coefficient matrix and assume the general case where $||B|| > 1$, we need to solve the following equation to get the inverse of B:

$$Bx = I$$

1. At the first step, we split B in the form:

$$B = \hat{B} - C$$

$\hat{B}$ is a matrix that the elements that are in the principal diagonal is defined as $\hat{b_{ii}} = b_{ii} + \alpha_i ||B||$. α_i is a parameter defined by the user for every $i = 1, 2, \ldots, n$. For the simplicity of the algorithm it is often easier to fix α a number larger than 1.

2. Generate the inverse of $\hat{B}$ by:

$$m_{rr'}^{(-1)} \approx \frac{1}{N} \sum_{s=1}^{N} \left[\sum_{(j|s_j = r')} W_j \right]$$

$$W_j = \frac{a_{rs_1} a_{s_1 s_2} \ldots a_{s_{j-1} s_j}}{p_{rs_1} p_{s_1 s_2} \ldots p_{s_{j-1} s_j}}$$

$(j|s_j = r')$ means only W_j that $s_j = r'$ are included in the sum.
It's the MC method for inverting diagonally dominant matrices, and it will be introduced more detailed in the next part.

3. Generate the inverse of B iteratively from the inverse of $\hat{B}$. For $k = n-1, n-2, \ldots, 0$:

$$B_k^{-1} = B_{k+1}^{-1} + \frac{B_{k+1}^{-1} S_{k+1} B_{k+1}^{-1}}{1 - trace\left(B_{k+1}^{-1} S_{k+1}\right)}$$

Trace of a matrix is defined to be the sum of the elements on the main diagonal. $\hat{B^{-1}}$ is used for $\hat{B_n^{-1}}$, and S_i is all zero except for all the iith component, which is from $S = \hat{B} - B$. Iteratively, we can calculate the inverse of B which is $\hat{B_0^{-1}}$.

(3) Monte Carlo Algorithm
1. Algorithm
The critical part of this paper is changing the method of getting the inverse of $\hat{B}$ to a MC algorithm, it's called the

Monte Carlo algorithm for inverting diagonally dominant matrices.[4]
Consider the Markov chain given by:

$$s_0 \to s_1 \to \cdots \to s_k$$

where the $s_i, i = 1, 2, \ldots, k$, belongs to the state space $S = \{1, 2, \ldots, n\}$. Then for $\alpha, \beta \in S, p_0(\alpha) = p(s_0 = \alpha)$ is the probability that the Markov chain starts at state α and $p(s_{j+1} = \beta | s_j = \alpha) = p_{\alpha\beta}$ is the transition probability from state α to state β. The set of all probabilities $p_{\alpha\beta}$ defines a transition probability matrix $P = \{p_{\alpha\beta}\}_{\alpha,\beta=1}^{n}$.
The distribution $(p_1, \ldots, p_n)^t$ is called acceptable for a given vector g, and the distribution $p_{\alpha\beta}$ is called acceptable for matrix T. Then the transition weight is defined as

$$W_0 = 1$$

and

$$W_j = W_{j-1} \frac{t_{s_{j-1}s_j}}{p_{s_{j-1}s_j}}$$

for $j = 1, 2, \ldots, n$.
Consider the random variable $\theta[g] = \frac{g_{s_0}}{p_{s_0}} \sum_{i=1}^{\infty} W_i f_{s_i}$, the following notation is used for the partial sum:

$$\theta_i[g] = \frac{g_{s_0}}{p_{s_0}} \sum_{j=0}^{i} W_j f_{s_j}$$

$\theta_i[g]$ can be considered as an estimate of (g, x) for i sufficiently large. To find an arbitrary component of the solution, for example the rth component of x, it follows that

$$(g, x) = \sum_{\alpha=1}^{n} \left(e^{(r)}\right)_{\alpha} x_{\alpha} = x_r$$

The corresponding MC method is given by:

$$x_r = \frac{1}{N} \sum_{s=1}^{N} \theta_i \left[e^{(r)}\right]_s$$

where N is the number of chains and $\theta_i \left[e^{(r)}\right]_s$ is the approximate value of x_r in the sth chain. It means that using an MC method makes it possible to estimate elements of the solution vector.
A Monte Carlo matrix inversion is obtained in a similar way, which explained the two equations in step 2 of the previous chapter.
In the above part, since W_j is included only into the corresponding sum for $r' = 1, 2, \ldots, n$, then the same set of N chains can be used to compute a single row of the inverse matrix, which is one of the inherent properties of MC making them suitable for parallelisation.
The probable error of the method, is defined as

$$r_N = 0.6745\sqrt{\frac{D\theta}{N}}$$

$$P\left\{\overline{\theta} - E(\theta) < r_N\right\} \approx \frac{1}{2} \approx P\left\{\overline{\theta} - E(\theta) > r_N\right\}$$

If one has N independent realisations of random variable θ with mathematical expectation Eθ average $\overline{\theta}$.

2. pseudocode
Let B denotes $\hat{B}$ in the following algorithm.
i. Input matrix B and parameters ϵ and δ for the markov chain.
ii. Split B=$B_1 - B_2$, where $B_1 = diag(B)$ and $B_2 = B_1 - B$.
iii. Calculate matrix A=$B_1^{-1} B_2$.
Then compute $||A||$ and the number of Markov chains $N = \left(\frac{0.6745}{\varepsilon(1-\|A\|)}\right)^2$.
iv. Compute the probability matrix P.
v. Calculate matrix M, by MCMC on A and P.

```
For i = 1 to n
    For j = 1 to N
        1. set W0=1, point =1, and
       SUM[k]=1 if i=k else SUM[k]=0
        2. Select a nextpoint based on transition
        probabilities in P so that A[point][nextpoint]!=0
        3. compute Wj=
       W(j-1)*A[point][nextpoint]/P[point][nextpoint]
        4. set SUM[nextpoint]=SUM[nextpoint]+Wj
        5. if abs(Wj)>delta, set point=
        nextpoint and goto 2
    m(ik)=SUM[k]/N for k=1,2,...,n
```

vi. Compute the Monte Carlo inverse:

$$B^{-1} = MB_1^{-1}$$

Through the use of the enhancement the algorithm is able to generate rough inverses of input matrices efficiently.
Monte Carlo methods for matrix inversion that only require O(NL) steps to find a single element or a row of the inverse matrix. Here N is the number of Markov chains and L is an estimate of the chain length in the stochastic process. To find the inverse matrix, the complexity is O(nNL).

E. Experiments

1. Testing and result

The author take two matrices $Psmigr\_3$ and $Appu$ for testing. $Psmigr\_3$ is a 3140*3140 matrix with 543162 non-zero entries, depicting data from US inter-county migration. $Appu$ is a 14, 000x14, 000 matrix with 1, 853, 104 nonzeros from NASAs app benchmark set.These matrices are used as inputs to both the MSPAI and our Monte Carlo based application to compute preconditioners.
Numerical experiments have been executed on the MareNostrum supercomputer, which currently consists of 3056 compute nodes that are each equipped with 2 Intel Xeon 8-core processors, 64GB RAM and are connected via an InfiniBand FDR-10 communication network.
The following figures are the test result for MSPAI and MCSPAI, including running time and residuals. Remainder residual can be regarded as the error rate, the lower the better.

2. Evaluation

The obtained values for the residuals are quite different between two compared approaches. The MSPAI based deterministic algorithm produces the same result for all runs whereas the Monte Carlo approach, due to its stochastic nature, generates unique preconditioners of a varying quality within the same preconfigured range every time they are

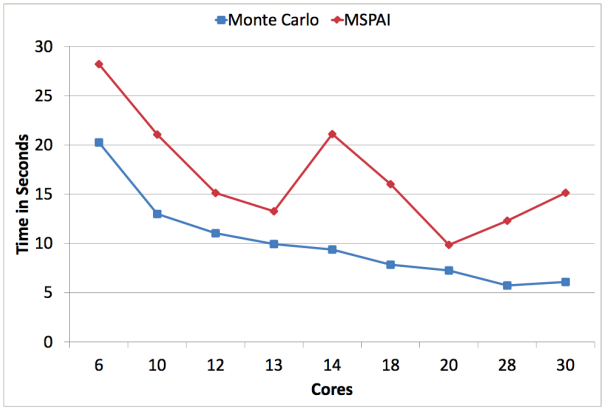


Fig. 1: Run Times for preconditioning $Psmigr\_3$

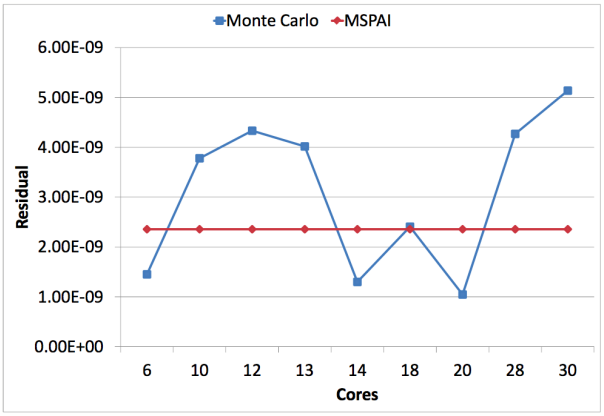


Fig. 2: Residuals for preconditioning $Psmigr\_3$

recomputed. This is easy to understand, the MCMC method is randomized, so the performance is different every time it runs.

From the results it can be seen that our proposed stochastic Monte Carlo approach is able to generate preconditioners of a good quality and within the same margin of error as the deterministic approach taken by MSPAI.

When comparing the time needed to compute a preconditioner it is noticeable in the case of $Psmigr\_3$ that both algorithms

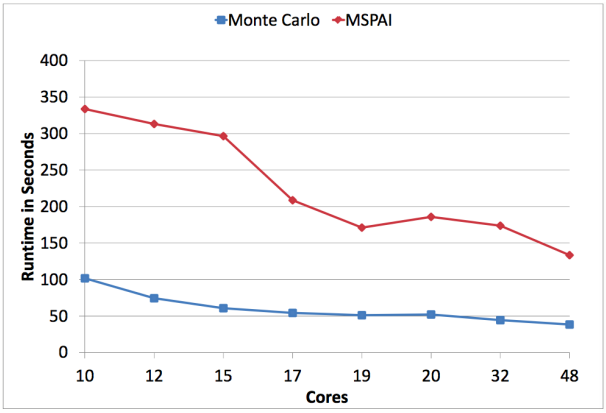


Fig. 3: Run Times for preconditioning Appu

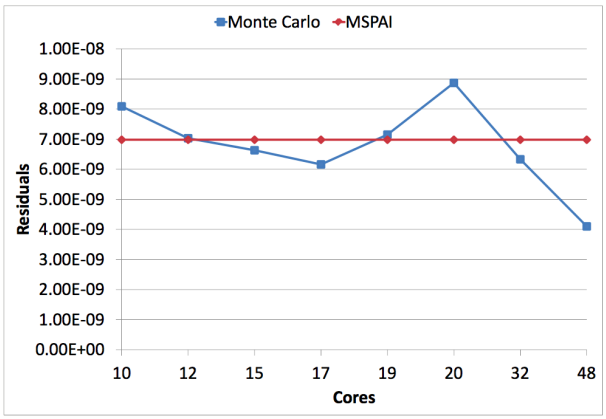


Fig. 4: Residuals for preconditioning Appu

follow roughly the same scaling pattern. The Monte Carlo approach is significantly faster in computing, especially in the case of fewer processors used.

When considering the larger test case of the $Appu$ matrix, the Monte Carlo algorithm outperforms the MSPAI approach quite significantly. It means the MC method has a obvious advantage when the matrix grows larger.

F. Conclusion

The algorithm the arthor propose is able to generate good quality preconditioners by generating a rough inverse using Markov Chain Monte Carlo methods. With an increased size of the input data, handling this information in the main memory of a computer will be a challenge, but with the inherent parallelism in Monte Carlo methods in mind, a parallel data replication approach for even larger matrices is worth investigating.

We think a random algorithm like MCMC is suitable and efficient for these kind of problems. We don't need to get a totally accurate solution, the goal is just to roughly estimate a value, the speed is the thing we follow with interest.

VI. Application2: Finding minimum node separators: A Markov chain Monte Carlo method

A. Introduction

In networked systems such as communication networks or power grids, graph separation from node failures can damage the overall operation severely. One of the most important goals of network attackers is thus to separate nodes so that the sizes of connected components become small.

In this work, we consider the problem of finding a minimum α-separator, that partitions the graph into connected components of sizes at most n,where n is the number of nodes

B. related work

the author classify previous work into several categories and summarize them as three parts:

1.Network vulnerability and reliability

there is a famous paper by Paolo et al. analyzing the size of the largest component after node removals and the ability to resist filures depending on the type of attacks.Other studies conclude providing a quantitative assessment of how failures affect the network structure.We won't give more details on their work since it is kind of irrelevant to our subject here.

2.Deterministic approaches and inapproximability of the α -separator problem

some studies have developed some polynomial-time algorithms for special topologies(trees and cycles).For example,Shen et al. also developed polynomial-time dynamic programming algorithms on tree structures and series-parallel graphs.Nevertheless, we still know little of general topologies.

3.Related problems of α-separator problem

the α-separator problem is a more generalized version of minimum vertex cover problem($\alpha = 1/n$) and minimum dissociation set problem($\alpha = 2/n$). Therefore,it's significant to develop an algorithm trying to solve α-separator problem.

C. System model

The author makes following notation and denotation to represent his model. Consider a simple graph G = (V, E) with V = n. Denote N (v) as a set of neighbors of a node $v \in V$. The author assumes that the attack cost is homogeneous across nodes. An α-separator W for $1/n \leq \alpha < 1$ is defined as a subset of nodes satisfing that the sizes of all components in the graph $G\backslash W$ is equal to or small than αn.The author calls this an α-separating condition. $G\backslash W = (V\backslash W, E(V\backslash W))$is obtained after removing the nodes in W and all their incident edges, where $E(V\backslash W) = \{(i,j) \in E | i \in V\backslash W, j \in V\backslash W\}$.

Here give the definition of α-sepatator problem for $1/n \leq \alpha < 1$ to find a minimum α-separator as follows:

$$\min_{W \subseteq V} |W| \text{ subject to } f(G \ \ W) \leq \alpha n$$

$f(G)$: the size of the largest connected component in G.

Make a notation $m = \lfloor \alpha n \rfloor$. We can prove that there is a polynomial reduction from minimum vertex cover problem to the α-separator problem,which means α-separator problem is NP-hard.The proof is as follows(the proof is given by [6]):

Construction:

denote: G = (V, E) is the graph we seeks to find a minimum vertex cover.

Build a new graph $G' = (V', E')$ by m duplications of each vertex $v \in V$. Replace all vertices $v \in V$ with m vertices $v_1, \cdots, v_m$. $(v_i, v_j) \in E'$ for all $1 \leq i, j \leq m$ and $(u_i, v_j) \in E'$ for all $1 \leq i, j \leq m$ when $(u, v) \in E$ for $u, v \in V$.

A minimum vertex cover $S \subseteq V$ directly leads to an α-separator that contains m duplicate vertices of each vertex of S. For a minimum α-separator of G', there is at least one u_l that belongs to the α -separator if two adjacent vertices u_i and v_j belong to the same connected component. Otherwise, the size of the connected component will be strictly greater than m, because all m duplicate copies of u and v_j are connected. By deleting v_j and adding u_l, we get a new minimum α-separator. By repeating this process, we can obtain a minimum α-separator where each connected component contains exactly m duplicate vertices of some vertices $v \in V$. We can form a minimum vertex cover that contains those vertices v in the minimum α-separator. Thus, the size of a minimum α-separator is $m|S|$ if and only if G has a vertex cover of size $|S|$. Since the minimum vertex cover problem is NP-hard, the α-separator problem is also NP-hard.

D. Random walk algorithm

After introducing some necessary notation and proof , let's talk about the essential part: some algorithms developed by the author of this paper.The first one is Random Walk Algorithm.

this algorithm takes one of three actions in every step:

(1)remove v from the attack set W

(2) add a vertex v in W

(3) stay at the current state

At the same time the author also applies a data sturcture and a corresponding update mechanism for the convenience of computing the sizes of connected components after each action.

Here is a short description of the algorithm: This algorithm runs over an underlying Markov chain.The state of the Markov chain represents one α-separator.the staionary distribution derived from the Markov chain we designed is as follows:

$$\pi(W) = \frac{\rho^{|W|}}{Z}$$

Z is a normalization constant

Here's the pseudo-code of the random walk algorithm(Fig.5)

$W = V; W_{\min} = W$
head(v) $= v, \forall v \in V$; cluster $(v) = \{v\}, \forall v \in V$
for step $= 1$ to numStep **do**
  Pick a vertex $v \in V$ uniformly at random
  if $v \in W$ **then**
    newClusterSize=1+$\sum_{i \in \cup_{j \in \mathcal{N}(v)\backslash W}} head(j)|cluster(j)|$
    if new Cluster Size $\leq \alpha n$ **then**
      $W = W\backslash\{v\}$; if $|W| < |W_{min}|, W_{\min} = W$
      cluster(v)= $\bigcup_{i \in \cup_{j \in \mathcal{N}(v)\backslash W}} head(j) cluster(i) \cup \{v\}$
      for $i \in$ cluster (v) **do**
        head $(i) = v$
      end for
    end if
  else if $v \notin W$, with probability ρ **then**
    $W = W \cup \{v\}$
    $\text{head}(i) = i$ for $i \in \mathcal{N}(v)\backslash W$
    for $i \in \mathcal{N}(v)\backslash W$ **do**
      if head $(i) = i$ **then**
        cluster $(i) \leftarrow graph$ Traverse $(i, G\backslash W)$
        head $(j) = i, \forall j \in$ cluster (i)
      end if
    end for
  end if
end for

the following figure(Fig.7) shows how the random walk algorithm workd:

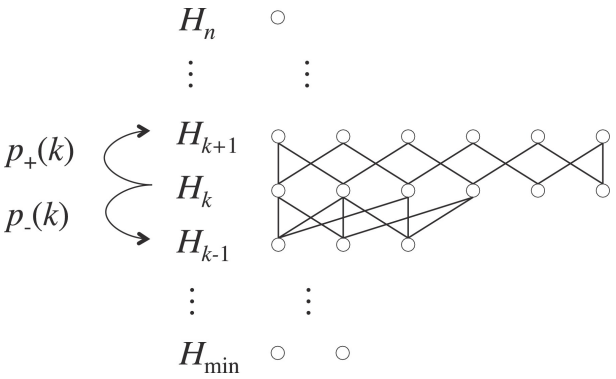


Fig. 5: Illustration of a hierarchical Markov chain of α-separators

In the beginning,the attact set W includes all vertices,V.W_{min} tracks the α-separator which is smallest and has been found.At each step, pick a random vertex ν(in line 4 of the pseudocode) if

(1)$\nu \in W$

the algorithm tryies to remove ν from the attack set W.

we first compute the size of the connected component that ν is associated with. Since this component is the only one that changes its size, if the size of this component is less than or equal to αn, the vertex addition of ν is acceptable. When the new connected component satisfies the condition, a vertex ν is removed from W. The vertex ν is assigned as a head vertex of this component and the set of associated vertices (i.e., cluster) is updated accordingly.

(2)$\nu \notin W$

the algorithm tries to add ν in W.

If $\nu \notin W$, the vertex ν is added to W with probability ρ. After ν is included in W, the connected component that ν was associated with can be divided into multiple components. To track this, we need to traverse from each neighboring vertex of ν, until all the neighboring vertices are visited (lines 16–22).

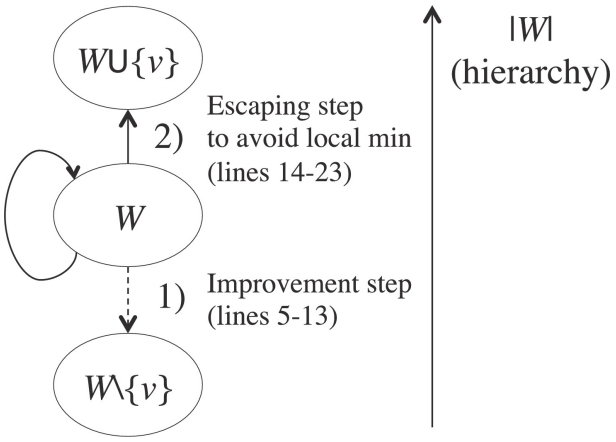
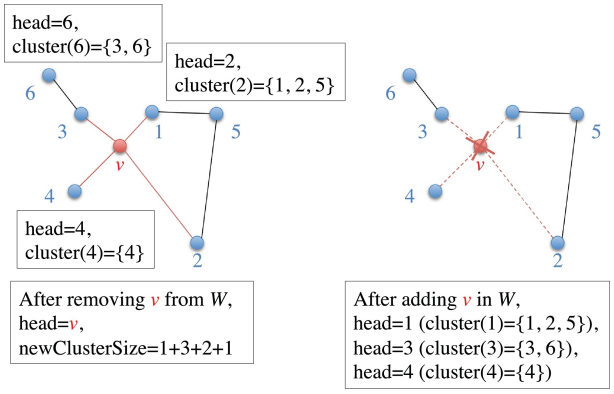


Fig. 6: transitions in the algorithm

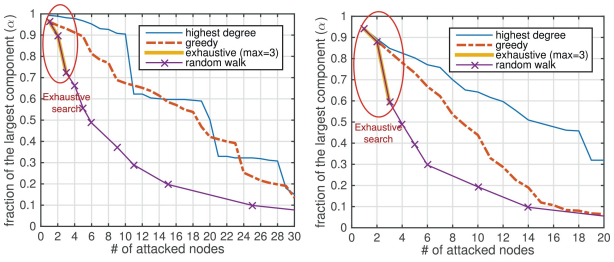
Function graphTraverse$(i, G\backslash W)$ traverses the graph $G\backslash W$ starting from node i.It will return the connected component (i.e., cluster) of node i. Pay attention to the fact that if some neighboring nodes are connected to the node i (e.g., nodes 2 and 1 in Fig. 3 (b)), we will update the head node of these nodes as i in line 20. We can use DFS(depth-first search) or BFS(Breadth-first search) to implement it.The complexity of DFS or BFS is $O(V + E)$. The number of graph traversals is at most the number of neighboring nodes, which is $O(n)$. The complexity in vertex removal from W is $O(|V|)$ and complexity in vertex addition to W is $O(V + E)$.



(a) Removing v from W (b) Adding v in W

Fig. 7: An example of vertex removal and addition to W adn updates of heads and clusters

In conclusion, the overall complexity has relation with the number of steps to run.which means we can not give a fixed complexity.Here's the complexity : $O(|V|^2 + |V||E|) \times numStep$, or $O(n^3) \times numStep$ (numStep : the number of steps to run).



(a) In US fiber networks. (b) In Italian power grid.

Fig. 8: Comparison with other algorithm

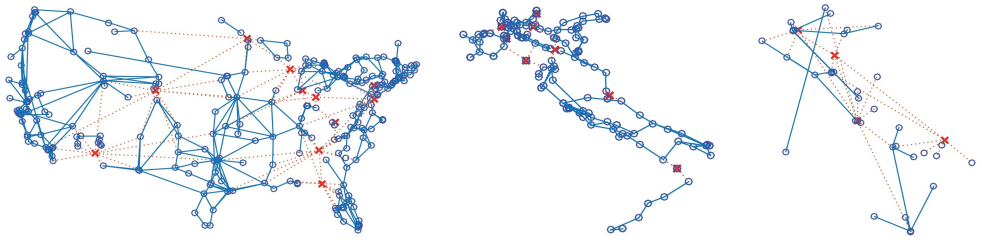
E. Simulation results

(1)Comparison with other algorithms In this paper the author compare his algorithm with exhaustive search and two simple heuristic algorithms: greedy and highest-degree-first. The algorithm is tested on various real and generated topologies:US fiber networks, Italian power grid and Internet network.(Fig.9)

(1)the exhaustive search

it requires computing the sizes of components for each subset, which takes $O(n^3)$ computations. The number of subsets with m nodes is $O(n^m)$. Finding a minimum takes $O(n^m)$ computations. Thus, the exhaustive search for subsets with m nodes takes $O(n^{m+3})$ computations.

(2) highest-degree first algorithm



(a) US fiber networks. (b) Italian power grid. (c) Italian Internet networks.

Fig. 9: the networks the author uses

short description of this algorithm:Attack vertices are chosen in the order of their node degree, i.e., the number of neighboring vertices. Break the tie randomly.

In the beginning,it sorts the vertices by their degree which only takes $O(nlog(n))$ computations. In order to obtain the sizes of the components, highest-degree first algorithm runs a graph traverse algorithm no more than n times. Therefore , the complexity : $O(n^3)$.

(3)greedy algorithm

short description of this algorithm: In each iteration, for each candidate vertex that is not in W, it computes the size of the largest connected component after adding the vertex. Then, the vertex with the largest marginal decrease is chosen. Break the tie randomly.

in each iteration, the greedy algorithm sorts the vertices taking $O(n^2log(n))$ computations, but the complexity is still $O(n^3)$ from the graph traverses.

In conclusion, the highest-degree-first and greedy algorithms perform much better than our algorithm in terms of time complexity(with $O(n^3)$ iterations).But their performance is much worse than our random walk algorithm in terms of results. The exhaustive search is optimal.However, it takes a really long time if m is not small.

VII. Conclusion

From what we discuss above, we can draw conclusions:

1) MCMC is a power tool, and it is based on Bayesian statistics. It can be applied to solve a large variety of data analysis problem.
2) As a standard methodology in MCMC, Metropolis Hastings Algorithm is a miraculously simple way to construct a Markov chain with P as its stationary distribution P. Beginning with a provisional Markov chain C_1 satisfying only some minimal requirements, it turns out to be very easy to modify C_1, making it into another chain C_2.
3) Gibbs sampling is a derivative of Metropolis Hastings Algorithm, which is pretty powerful for related variables.
4) For Metropolis Hastings Algorithm and its derivatives, we usually have a dilemma, for exploration and exploitation. To solve it, we usually use a dynamic epsilon and annealing Algorithm.

Acknowledgment

[1]Ziyu Shao

References

[1] Straßburg J, Alexandrov V. Enhancing Monte Carlo preconditioning methods for matrix computations[J]. Procedia Computer Science, 2014, 29: 1580-1589.
[2] A study of computational complexity of algorithms for numerical methods.University of Rajasthan, 2014.
[3] T. Barnard, Stephen and Grote, Marcus. (1999). A Block Version of the SPAI Preconditioner.
[4] Straßburg, Janko, Alexandrov, Vassil. (2013). On scalability behaviour of Monte Carlo sparse approximate inverse for matrix computations. 10.1145/2530268.2530274.
[5] Lee J, Kwak J, Lee H W, et al. Finding Minimum Node Separators: A Markov Chain Monte Carlo Method[J]. Reliability Engineering and System Safety, 2018.
[6] Sidi MAM. K-separator problem, Ph.D. thesis. Evry, Institut National des Télécommunications; 2014.
[7] Levin DA, Peres Y, Wilmer EL. Markov chains and mixing times. American Mathematical Society; 2009.