SAA and SAMC for minimum graph bisection

Tyler Grimes

December 12, 2016

1 Introduction

Let $G = (V, E)$ be an undirected graph with vertices $V = \{1, ..., n\}$ having weights w_i with $0 < w_i \leq p$ for $i = 1, ..., n$ and some positive constant p, and $m = |E|$ edges with weights $c_{i,j} \geq 0$ for $i, j = 1, ..., n$ where $c_{i,j} = 0$ indicates that there is no edge between vertices i and j.

There are many variations to the graph partition problem. In general, consider the problem of partitioning the vertices into k disjoint *admissible* sets, $A_1, ..., A_k$, with $V = A_1 \cup ... \cup A_k$. A vertex set A is admissible if $size(A) = \sum_{j \in A} w_j \leq p$. A partition is admissible if each set in the partition is admissible.

A partition generates a graph cut after removing all edges between vertices in different subsets. That is, for each $i \in V$, set $c_{i,j} = 0$ if i and j are not in the same subset. For a given partition $A_1, ..., A_k$, we can denote the set of removed edges by $R_{A_1,...,A_k} = \{\{i, j\} : \{i, j\} \not\subset A_p \text{ for any } p\} \subseteq E$, or similarly, the set of retained edges $K_{A_1,...,A_k} = \{\{i, j\} : \{i, j\} \subset A_p \text{ for some } p\} \subseteq E$. The *external cost* of a partition is defined as $E(A_1, ..., A_k) = \sum_{\{i,j\} \in R} c_{i,j}$, that is, the sum of all removed edge weights. The *internal cost* of a partition is defined as $I(A_1, ..., A_k) = \sum_{\{i,j\} \in K} c_{i,j}$, that is, the sum of all retained edge weights.

The optimization problem is to find an admissible partition that has minimal external cost. Note that the solution to this problem need not be unique. In addition, since the total weight of all edges is constant, this problem is equivalent to finding an admissible partition that has maximal internal cost.

1.1 Brute force search and computational complexity

An exact solution can be found by a brute force search. This procedure would entail iterating through all possible partitions and calculating the external cost of each. If an admissible partition is found that has the lowest cost observed up to that point, the solution is updated to be this partition. This method is simple, but is intractible even for small graphs. Consider the graph of n vertices with all vertices having weight $w_i = 1$, and we wish to find a partition of k subsets each of size p where $kp = n$. There are $\binom{n}{p}$ ways to choose the first subset, $\binom{n-p}{p}$ ways to choose the second, and so on. We don't care about the $k!$ ways of ording the subsets, so the total number of partitions is

$$\frac{1}{k!}\binom{n}{p}\binom{n-p}{p}...\binom{2p}{p}\binom{p}{p}.$$

With just 30 vertices and $k = 2$ partitions ($p = 15$), there are on the order of 10^7 partitions. With 100 vertices, that number grows to over 10^28. It turns out that this problem is NP-

complete (Gary & Johnson, 1979), and so an efficient approximation method is desirable.

1.2 Heuristics

Instead of using a method that guaruntees to find the global optimum, heuristic techniques are employed which run faster and provide good, but not always optimal, solutions. One class of heuristics iteratively improves a partition by swapping or moving verticies between partitions one at a time. This class is referred to as move-based algorithms or local searches. One such method is the Kernighan-Lin (KL) algorithm (Kernighan & Lin, 1970). However, these approaches can get stuck at a local optimum. This leads to the second class of heuristics, namely, stochastic methods, which can avoid local trips. An early application of simmulated annealing was used in this context (Kirkpatrick, 1984).

To handle very large graphs, multilevel approaches have been proposed (Karypis & Kumar, 1995). However, these methods still rely on local search and stochastic methods to refine their solutions.

The remaining sections provide a brief summary of the KL and simulated annealing algorithms, followed by two proposed heuristics based on SAA and SAMC. Our attention will be restricted to the minimum graph bisection problem (that is, where $k = 2$ and $kn = p$). A small simulation is run for a cursory investigation into the performance of these four methods.

2 Kernighan-Lin (KL) algorithm

The KL algorithm (Kernighan & Lin, 1970) is a heuristic that runs in $O(n^3)$ time. The procedure starts with a random partition. It then iteratively finds a pair of verticies in each subset that gives the largest decrease (or smallest increase) in cut size if the two are swapped. This pair is then locked, and the procedure repeats until all verticies are locked. At the end, the subset of swaps that gives the total largest decrease in cut size is chosen, and the swaps are performed. If none of the swaps reduce the cut size, then the initial partition is left unchanged. The resulting partition can then be fed back into the procedure, repeating the algorithm until the cut size can no longer be decreased (a local optimal is found).

This algorithm can be extended to handle k-way partitions, unequal size partitions, and unequally weighted verticies, however the addition of dummy verticies is required for the latter two.

KL algorithm:

Parameters: W - a $2n \times 2n$ adjacency matrix.

A - optional: start from an initial partition A and $\bar{A}$. A must be of length n.

(1) If A is not provided, initialize A and $\bar{A}$ by randomly assigning n verticies to A and the rest to $\bar{A}$.
(2) For each vertex c, initialize the external cost $E_c = \sum_{v \in \bar{C}} W_{cv}$, the internal cost $I_c = \sum_{v \in C} W_{cv}$, and the D-value (cost reduction for moving c) $D_c = E_c - I_c$, where C is the partition subset containing c.
(3) For i in 1 to n:

(3.1) Compute the cost reduction $g_{aa'} = D_a + D_{a'} - 2W_{aa'}$ of each pair of unlocked verticies $a \in A$ and $a' \in \bar{A}$.
(3.2) Find the pair (a, a') that has maximal cost reduction $g_{aa'}$.
(3.3) Set $(a_i, a'_i) = (a, a')$, $\hat{g}_i = g_{aa'}$ and mark a and a' as locked.
(3.4) Update $D'_x = D_x + 2W_{xa} - 2W_{xa'}$ and $D'_y = D_y + 2W_{ya} - 2W_{ya'}$ for all unlocked $x \in A$ and unlocked $y \in \bar{A}$.

(4) Find k such that $G_k = \sum_{i=1}^{k} \hat{g}_i$ is maximized.
(5) If $G_k > 0$, swap each pair (a_i, a'_i) for $i = 1, ..., k$.
(6) Return A and $\bar{A}$

---

3 Simulated annealing (SA)

Simulated annealing was introduced by Kirkpatrick (1984) who also showed how it can be applied to the graph bisection problem. Johnson et al. (1989) reported on an extended empirical study comparing SA to KL. It was shown that SA performed better on sparse random graphs, but worse on graphs with geometric structure.

For the minimum graph bisection problem, a potential solution S forms a partition $\left\{S, \bar{S}\right\}$, with both subsets having equal size. A neighbor S' of S is a partition obtained by swapping one pair of verticies (a, b), $a \in S$ and $b \in \bar{S}$. The cost of a solution $cost(S) = \sum_{i \in S, j \in \bar{S}} W_{ij}$ is the external cost of the partition.

The minimization problem is to find

$$S = \min_{S \in \chi} cost(S), \tag{1}$$

where χ is the set of all possible bipartitions. This minimization problem is equivalent to sampling from the Boltzmann distribution

$$P_{\tau*}(S) \propto \exp\{-cost(S)/\tau*\} \tag{2}$$

as the temperature $\tau*$ is lowered toward zero on a logarithmic cooling schedule (Kirkpatrick et al., 1983). However, log cooling is usually too slow be of use, so we instead approximate the procedure by using a square-root cooling schedule.

SA algorithm:

Parameters: W - a $2n \times 2n$ adjacency matrix.

τ_0 - the starting temperature (must be strictly positive).

(1) Initialize a solution S by randomly assigning n verticies to A and the rest to $\bar{A}$.
(2) Set the initial temperature $\tau = \tau_0$.
(3) While not yet frozen:

(3.1) Pick a random neighbor S'.
(3.2) Compute $\Delta = cost(S) - cost(S')$.
(3.3) Set $S_{t+1} = S'$ with probability $min(1, \exp(-\Delta/\tau))$. Else $S_{t+1} = S_t$.
(3.4) Cool the temperature $\tau_{t+1} = \tau_0/\sqrt{\text{t}}$.

(4) Return the S with the lowest cost.

The procedure should be iterated until convergence; however, determining whether the procedure has converged is difficult. Instead, the procedure can be run for a fixed number of iterations, or until the temperature is lowered below some threshold.

4 Simulated stochastic annealing approximation (SAA)

The SAA algorithm (Liang et al., 2014) can be applied to the graph partitioning problem. This is a direct extention of SA, only now a record is maintained of where we have been in the sample space. This history is taken into account when deciding on an uphill move, making the move more likely if the region has not been explored in a while. The sample space is partitioned into m subsets

$$\begin{aligned} E_1 &= \{S : cost(S) \leq c_1\}, ..., \\ E_{m-1} &= \{S : c_{m-2} < cost(S) \leq c_{m-2}\}, \\ E_m &= \{S : cost(S) > c_{m-1}\}, \end{aligned}$$

where $c_1 < ... < c_{m-1}$ are user specified values. The Boltzmann distribution from (2) is adjusted to

$$P_{\theta_t,\tau_{t+1}}(S) \propto \sum_{i=1}^{m} \exp\left\{-cost(S)/\tau_{t+1} - \theta_t^{(i)}\right\} I(S \in E_i), \tag{3}$$

where $I(\cdot)$ is the indicator function, θ_t is a vector of length m with θ_1 initialized to zero and $\theta_{t+1} = \theta_t + \gamma_{t+1} H_{\tau_{t+1}}(\theta_t, S_{t+1})$, $\gamma_t = \left(\frac{T_0}{max(t,T_0)}\right)$ is the gain factor sequence, $\tau_t = \tau_0/\sqrt{t}$ is the temperature sequence, $H_{\tau_{t+1}}(\theta, S) = e_{J(S)} - \pi$, $e_{J(S)}$ is a vector of $m-1$ zeroes and a 1 at the $J(S)$'th index, $J(S)$ is the index of E that S is in, and $\pi_i = \frac{\exp(-\eta(i-1))}{\sum_{k=1}^{m} \exp(-\eta(i-1))}$ is the desired sampling distribution over the partition ($\eta = 0.05$ will be used).

SAA algorithm:

Parameters: W - a $2n \times 2n$ adjacency matrix.

τ_0 - the starting temperature (must be strictly positive).

E - the partition of the sample space.

(1) Initialize a solution S_1 by randomly assigning n verticies to A and the rest to $\bar{A}$.
(2) Initialize $t = 1$, $\tau_t = \tau_0$, $\theta_t = 0$
(3) While not yet frozen:
  (3.1) Pick a random neighbor S'
  (3.2) Compute $\Delta = cost(S) - cost(S')$.
  (3.3) Set $S_{t+1} = S'$ with probability $\min\left(1, \exp(-\Delta/\tau + \theta_{J(S)} - \theta_{J(S')})\right)$. Else set $S_{t+1} = S_t$
  (3.4) Update $\theta_{t+1} = \theta_t + \gamma_{t+1}(e_{J(S_{t+1})} - \pi))$

(3.5) Cool the temperature $\tau_{t+1} = \tau_0/\sqrt{t+1}$.
(3.6) Set $t = t + 1$

(4) Return the S with the lowest cost.

---

5 Stochastic approximation Monte Carlo (SAMC)

The SAMC algorithm (Liang et al., 2007) can also be applied to the graph partitioning problem. This is a sampling procedure that partitions the sampling space in the same way as SAA. However, there is no annealing here. The target distribution is,

$$P_{\theta_t}(S) \propto \sum_{i=1}^{m} \exp\left\{-cost(S) - \theta_t^{(i)}\right\} I(S \in E_i). \tag{4}$$

Recall that θ is updated by $\theta_{t+1} = \theta_t + \gamma_{t+1}(e_{J(S_{t+1})} - \pi))$. If π is uniform, then SAMC performs a random walk along the subregions. By setting $\pi_i = \frac{\exp(-0.05(i-1))}{\sum_{k=1}^{m}\exp(-0.05(i-1))}$, the sampler is encouraged to spend more time in the lower subregions. Although there is no guaruntee of converging to a global minimum, SAMC is immune to local traps and will continuously explore the low subregions.

SAMC algorithm:

Parameters: W - a $2n \times 2n$ adjacency matrix.

E - the partition of the sample space.

$ITER$ - the number of iterations to perform.

(1) Initialize a solution S_1 by randomly assigning n verticies to A and the rest to $\bar{A}$.
(2) Initialize $t = 1$ and $\theta_t = 0$
(3) For ITER iterations:

(3.1) Pick a random neighbor S'
(3.2) Compute $\Delta = cost(S) - cost(S')$.
(3.3) Set $S_{t+1} = S'$ with probability $\min\left(1, \exp(-\Delta + \theta_{J(S)} - \theta_{J(S')})\right)$. Else set $S_{t+1} = S_t$
(3.4) Update $\theta_{t+1} = \theta_t + \gamma_{t+1}(e_{J(S_{t+1})} - \pi))$
(3.5) Set $t = t + 1$

(4) Return the S with the lowest cost.

---

6 Simulation

A small simulation study is performed to compare the four procedures. The first task is to generate a graph. The structure of the graph should be given consideration, as each method will perform differently depending on the properties of the graph. However, we only consider sparce graphs here, with each node being connected to 5% of the other nodes on average. After the edges are determined, the edge weights are each generated independently from a

uniform distribution. Graphs of size $n = 100, 250, 500, 750, 1000, 2000, 5000$, and 10000 were considered.

The initial temperature for the SA and SAA procedures are set at $\tau_0 = 100$. For SAA and SAMC, a partition size of $m = 101$ is used. To determine the upper and lower thresholds for the partition, the minimum cost c^* from the first 1000 iterations of SA is used. Set $c_1 = 0.8c^*$ and $c_{m-1} = 1.5c^*$. The T_0 used by the gain factor is set to 5000.

The KL algorithm is applied iteratively by passing back in the partition it produces until it can no longer improve the partition. The SA, SAA, and SAMC methods are run for 10^5 iterations each. The results from one sample of each graph size are shown below.

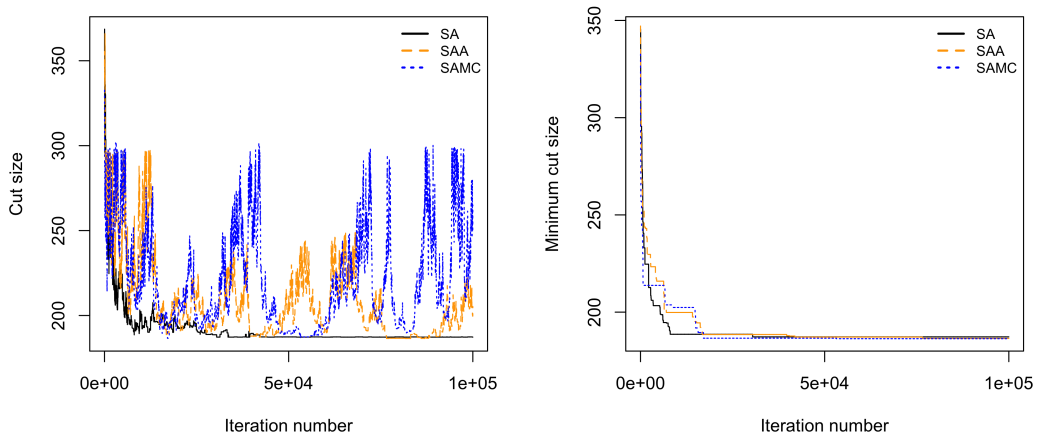

Figure 1: Comparison of SA, SAA, and SAMC for $n = 100$

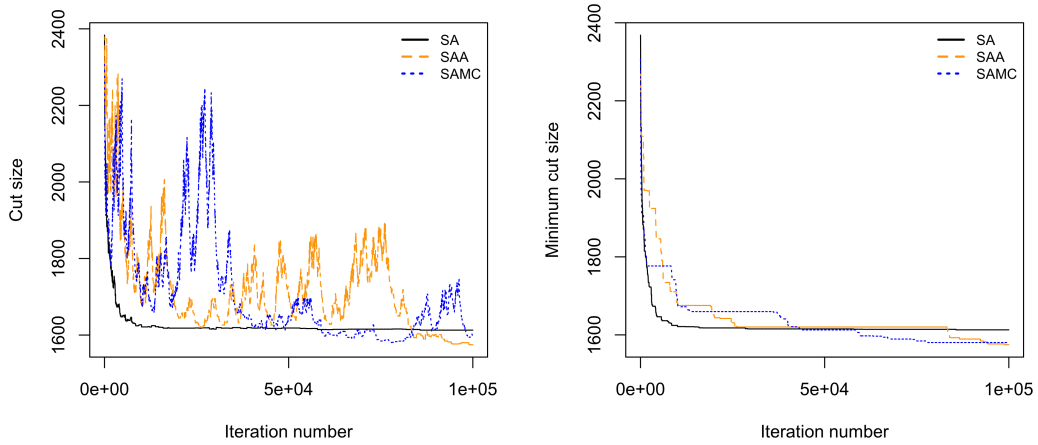

Figure 2: Comparison of SA, SAA, and SAMC for $n = 250$

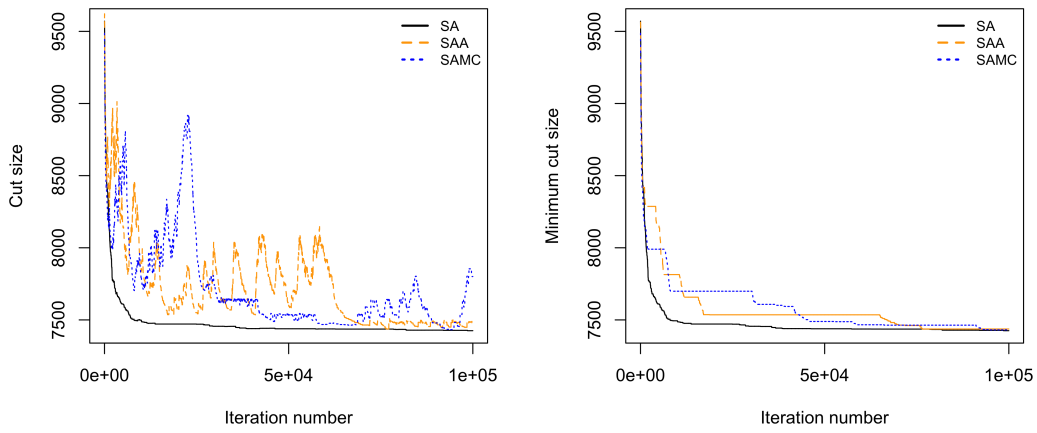

Figure 3: Comparison of SA, SAA, and SAMC for $n = 500$

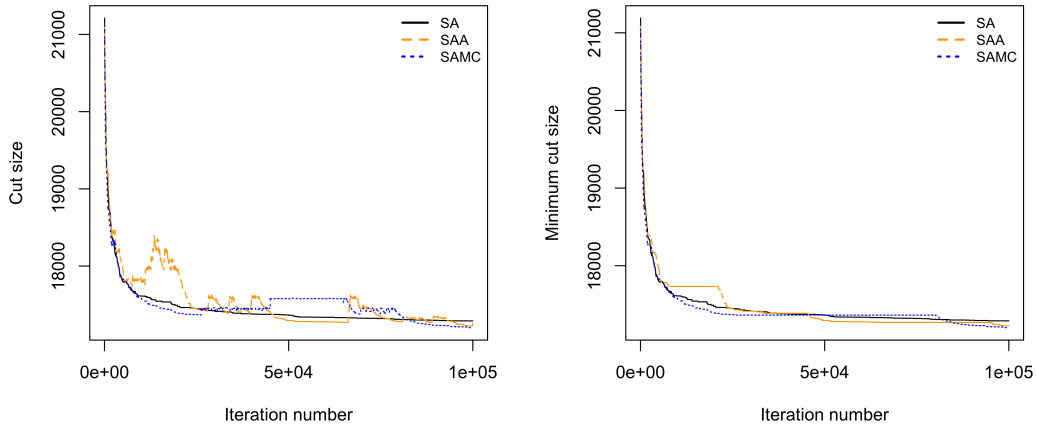

Figure 4: Comparison of SA, SAA, and SAMC for $n = 750$

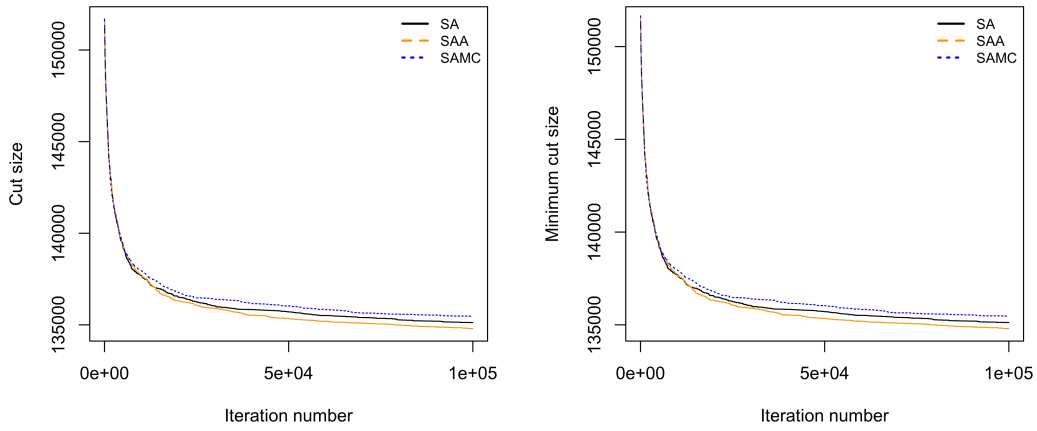

Figure 5: Comparison of SA, SAA, and SAMC for $n = 2000$

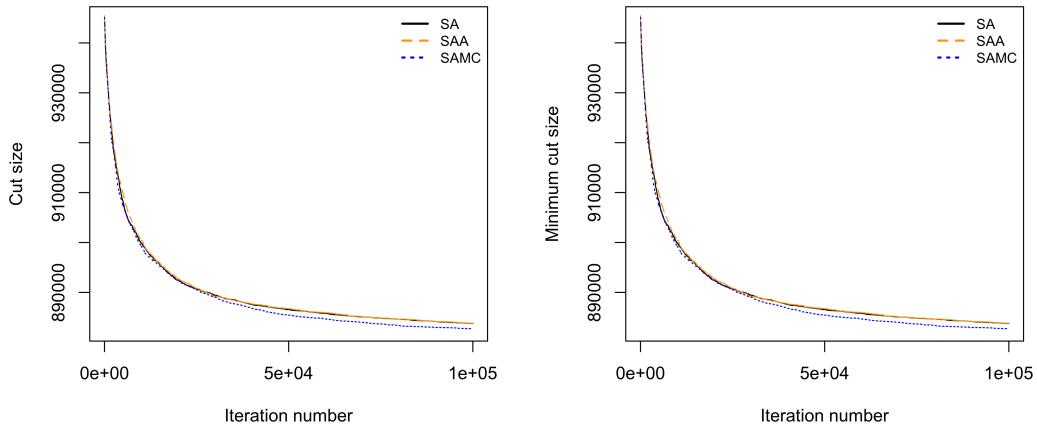

Figure 6: Comparison of SA, SAA, and SAMC for $n = 5000$

	100	250	500	750	1000	2000	5000	10000
KL	188	1622	7390	17233	31909	134097	NA	NA
SA	187	1613	7425	17290	32090	135121	883773	3611748
SAA	186	1575	7437	17225	31971	134791	883830	3614319
SAMC	186	1580	7431	17202	31966	135473	882728	3614278

Figure 7: Minimum cut found by each procedure for graph size n.

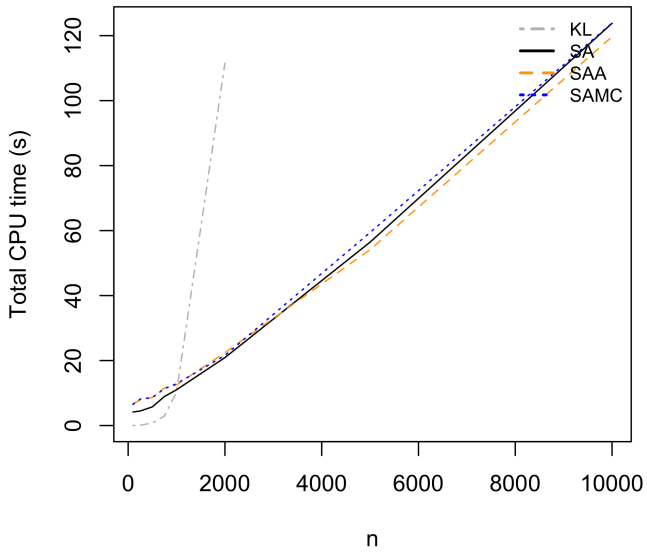

Figure 8: Total CPU time for graph size n. The KL algorithm was not run for $n > 2000$.

7 Discussion

For graphs with 1000 or fewer verticies, the KL algorithm is a very good heuristic to use; it is fast and often produces the best partition. However, because of its $O(n^3)$ complexity, KL slows down dramatically for graphs of size 2000 or larger. On the otherhand, the stochastic algorithms run in linear time and can easily handle larger graphs. The SAA and SAMC procedures have extra bookkeeping due to the partitioning of the sample space, so they tend to run slightly longer than SA. However, they also tend to find a minimum cut faster than SA. Regardless, more simulations are required before we can accurately compare SA, SAA, and SAMC.

The key property of SAMC is its ability to freely move around the sample space. This property is observed in figure 1 and figure 2. Both SAA and SAMC explore the sample space much more than SA, which allows them to not only avoid local traps but also find optimal points quickly. As the graph size increases they tend to explore less, but this may be due to the way the partition was chosen. Alternatively, since the minimum cut size increase with graph size, perhaps a rescaling of the edge weights could lead to better results. Either way, these results suggest that SAA and SAMC are likely improvements over SA.

Appendix A: R code

```
set.seed(1000)
library(gridExtra)

# ########################################################
```

```
#
# Auxillary functions.
#
# ####################################################
#Generate data set for t-test.
generate_data <- function(n = 100, weights = 2) {
  W <- matrix(0, nrow = n, ncol = n)
    for(i in 1:n) {
      m <- rbinom(1, n, 0.05) #Number of latent variables.
      latent_vars <- sample((1:n)[-i], m)
      vals <- runif(m)*weights
      W[i, latent_vars] <- W[i, latent_vars] + vals
      W[latent_vars, i] <- W[i, latent_vars] + vals
    }
}

# W: 2n by 2n adjacency matrix
# A: vector of size n containing the indicies in one subset.
# a: include if cost of only one node is desired.
external_cost <- function(W, A, a = NULL) {
  if(!is.null(a)) {
    return(sum(W[a, -A]))
  }
  c <- rep(0, nrow(W))
  c[A] <- apply(W[A, -A], 1, sum)
  c[-A] <- apply(W[-A, A], 1, sum)
  return(c)
}

# W: 2n by 2n adjacency matrix
# A: vector of size n containing the indicies in one subset.
# a: include if cost of only one node is desired.
internal_cost <- function(W, A, a = NULL) {
  if(!is.null(a)) {
    return(sum(W[a, A]))
  }
  c <- rep(0, nrow(W))
  c[A] <- apply(W[A, A], 1, sum)
  c[-A] <- apply(W[-A, -A], 1, sum)
  return(c)
}

#Swap two nodes between partitions A and -A.
move <- function(W, A) {
  B <- (1:(2*length(A)))[-A]
  a <- sample(A, 1)
  b <- sample(B, 1)
```

```
  delta_t <- internal_cost(W, A, a) +
    internal_cost(W, B, b) - external_cost(W, A, a) -
    external_cost(W, B, b) + 2*W[a, b]
  A_new <- A
  A_new[which(A == a)] <- b
  return(list(A_new = A_new, delta_t = delta_t))
}

cut_size <- function(W, A) {
  if(length(unique(A)) != length(A)) {
    stop("A should contain unique elements.")
  }
  return(sum(external_cost(W, A)[A]))
}

#Find which subset t belongs to in the partition.
get_partition_index<-function(t, E){
  m <- length(E)
  #First check edge cases.
  if(t < E[1]) {
    return(1)
  } else if (t >= E[m]) {
    return(m + 1)
  }

  #Binary search to find which subset contains t.
  binary_search <- function(t, indicies) {
    m <- length(indicies)
    if(m == 1) {
      if(t < E[indicies]) {
        return(indicies)
      } else {
        return(indicies + 1)
      }
    }
    i <- ceiling(m/2) + 1 #Start in the middle.
    if(t >= E[indicies[i - 1]] && t < E[indicies[i]]) {
      return(indicies[i])
    }
    if(t < E[indicies[i - 1]]) {
      return(binary_search(t, indicies[1:(i - 1)]))
    } else {
      return(binary_search(t, indicies[i:m]))
    }
  }

  return(binary_search(t, 2:m))
```

```
}



## Approximate bisection
# Code adapted from C.Ladroue at
#  https://www.r-bloggers.com/graph-bisection-in-r/
# ########################################################
#
# KL
#
# ########################################################
KL <- function(W, A = NULL){
  n <- nrow(W)
  m <- n/2

  # start off with a random partition
  if(is.null(A)) {
    A <- sample(1:n, n/2, replace=FALSE)
  }
  B <- (1:n)[-A]

  DA <- rowSums(W[A, B]) - rowSums(W[A, A]) + diag(W[A, A])
  DB <- rowSums(W[B, A]) - rowSums(W[B, B]) + diag(W[B, B])
  unmarkedA <- 1:m
  unmarkedB <- 1:m
  markedA <- rep(0,m)
  markedB <- rep(0,m)
  gains <- rep(0,m)
  for(k in 1:m) {
    dimension <- m+1-k
    fasterGain <- matrix(DA[unmarkedA], nrow=dimension,
                         ncol=dimension,byrow=FALSE) +
      matrix(DB[unmarkedB],nrow=dimension, ncol=dimension,
             byrow=TRUE) -
      2*W[A[unmarkedA], B[unmarkedB]]

    # mark the best pair
    best <- arrayInd(which.max(fasterGain),
                     .dim=c(dimension,dimension))
    besti <- unmarkedA[best[1]]
    bestj <- unmarkedB[best[2]]
    bestGain <- fasterGain[best]
    markedA[k <- unmarkedA[best[1]]
    markedB[k <- unmarkedB[best[2]]
    unmarkedA <- unmarkedA[-best[1]]
    unmarkedB <- unmarkedB[-best[2]]
```

```
    # record gain
    gains[k <- bestGain

    # update D for unmarked indices
    DA[unmarkedA <- DA[unmarkedA] +
         2*W[A[unmarkedA], A[besti]] -
         2*W[A[unmarkedA], B[bestj]]
    DB[unmarkedB <- DB[unmarkedB] +
         2*W[B[unmarkedB], B[bestj]] -
         2*W[B[unmarkedB], A[besti]]
  }

  gains <- cumsum(gains)
  bestPartition <- which.max(gains)
  maxGain <- gains[bestPartition]

  if(maxGain>0) {
    # swap best pairs
    A1 <- c(A[-markedA[1:bestPartition]],
            B[markedB[1:bestPartition]])
    B1 <- c(B[-markedB[1:bestPartition]],
            A[markedA[1:bestPartition]])
    A <- A1
    B <- B1
  }

  list(A = A, B = B, max_gain = maxGain)
}


# ######################################################
#
# Simulated annealing.
#
# ######################################################
#Simulated Annealing algorithm.
SA <- function(W, A = NULL, initial_t = 50, MAX_ITER = 10^5) {
  if(nrow(W) %% 2 != 0) {
    stop("W should have an even number of nodes.")
  }
  n <- nrow(W)/2
  if(is.null(A)) {
    A <- sample(1:(2*n), n)
  }

  #Compute cut size of initial partition.
  t <- cut_size(W, A)
```

```
  temp <- initial_t

  #Maintain history of minimum cut sizes.
  t_history <- rep(0, MAX_ITER/10)
  min_t_history <- rep(0, MAX_ITER/10)
  t_history[1] <- t
  min_t_history[1] <- t
  min_t <- t
  min_A <- A

  #Begin SA:
  for(i in 1:MAX_ITER) {
    changes <- move(W, A)
    A_new <- changes$A_new
    delta_t <- changes$delta_t
    t_new <- t + delta_t

    #Determine if new tour should be accepted.
    if(delta_t < 0) {
      A <- A_new
      t <- t_new
    } else if(exp(-delta_t/temp) > runif(1, 0, 1)) {
      A <- A_new
      t <- t_new
    }

    #Cool the temperature (square root cooling rate).
    temp <- initial_t/sqrt(i)

    #Update history of cut sizes.
    if(t_new < min_t) {
      min_A <- A
      min_t <- t_new
    }
    if(i %% 10 == 0) {
      t_history[i/10] <- t
      min_t_history[i/10] <- min_t
    }
  }

  return(list(A = min_A, t_history = t_history,
              min_t_history = min_t_history,
              iterations = MAX_ITER))
}


# ##################################################
```

```
#
# Stochastic approximation annealing.
#
# ####################################################
SAA <- function(W, A = NULL, initial_t = 60, MAX_ITER = 10^4,
                E = seq(10000, 12000, length.out = 100),
                PI = NULL, t0 = 1000) {

  if(nrow(W) %% 2 != 0) {
    stop("W should have an even number of nodes.")
  }
  n <- nrow(W)/2
  if(is.null(A)) {
    A <- sample(1:(2*n), n)
  }
  m <- length(E)

  # Define initial values for SAA
  PI <- exp(-0.05*(1:(m + 1) - 1)); PI <- PI / sum(PI)
  gamma <- t0/(pmax(t0, 1:MAX_ITER))
  theta <- rep(0, m + 1)

  #Compute cut size of initial partition.
  t <- cut_size(W, A)

  #Find E index of current cut size.
  index <- get_partition_index(t, E)

  #Initialize the temperature
  temp <- initial_t

  #Maintain history of minimum cut sizes.
  t_history <- rep(0, MAX_ITER/10)
  min_t_history <- rep(0, MAX_ITER/10)
  t_history[1] <- t
  min_t_history[1] <- t
  min_t <- t
  min_A <- A

  #Begin SAA:
  for(i in 1:MAX_ITER) {
    #Sample a new tour.
    changes <- move(W, A)
    A_new <-changes$A_new
    delta_t <- changes$delta_t
    t_new <- t + delta_t
```

```
    #Determine if new tour should be accepted.
    if(delta_t < 0) {
      A <- A_new
      t <- t_new
      index <- get_partition_index(t, E)
    } else {
      new_index <- get_partition_index(t + delta_t, E)
      r <- exp(-delta_t/temp + theta[index] - theta[new_index])
      u <- runif(1, 0, 1)
      if(r > u){
        A <- A_new
        t <- t_new
        index <- new_index
      }
    }

    #Update theta.
    indicator <- rep(0, m + 1)
    indicator[index] <- 1
    theta <- theta + gamma[i]*(indicator - PI)

    #Cool the temperature (square root cooling rate).
    temp <- initial_t/sqrt(i)

    #Update history of cut sizes.
    if(t_new < min_t) {
      min_A <- A
      min_t <- t_new
    }
    if(i %% 10 == 0) {
      t_history[i/10] <- t
      min_t_history[i/10] <- min_t
    }
  }

  return(list(A = min_A, t_history = t_history,
              min_t_history = min_t_history,
              iterations = MAX_ITER))
}


# ######################################################
#
# Stochastic approximation Monte Carlo
#
# ######################################################
SAMC <- function(W, A = NULL, MAX_ITER = 10^5,
```

```
                E = seq(1100, 1400, length.out = 100),
                PI = NULL, t0 = 5000) {
if(nrow(W) %% 2 != 0) {
  stop("W should have an even number of nodes.")
}
n <- nrow(W)/2
if(is.null(A)) {
  A <- sample(1:(2*n), n)
}
m <- length(E)

#Initialize variables for SAMC.
PI <- exp(-0.05*(1:(m + 1) - 1)); PI <- PI / sum(PI)
gamma <- t0/(pmax(t0, 1:MAX_ITER))
theta <- rep(0, m + 1)

#Find the sampling partition index of the current cut size.
t <- cut_size(W, A)
partition_index <- get_partition_index(t, E)

#Maintain history of minimum cut sizes.
t_history <- rep(0, MAX_ITER/10)
min_t_history <- rep(0, MAX_ITER/10)
t_history[1] <- t
min_t_history[1] <- t
min_t <- t
min_A <- A

for(i in 2:MAX_ITER) {
  #Generate a permutation from the given sample.
  changes <- move(W, A)
  A_new <-changes$A_new
  delta_t <- changes$delta_t

  #Find the partition index of this t-statistic.
  t_new <- t + delta_t
  partition_index_new <- get_partition_index(t_new, E)

  #Compute the MH ratio.
  r <- exp(-delta_t + theta[partition_index] -
             theta[partition_index_new])

  if(r > runif(1)) {
    A <- A_new
    t <- t_new
    partition_index <- partition_index_new
  } else {
```

```
      #Do nothing.
    }

    #Update theta.
    indicator <- rep(0, m + 1)
    indicator[partition_index] <- 1
    theta <- theta + gamma[i]*(indicator - PI)

    #Update history of cut sizes.
    if(t_new < min_t) {
      min_A <- A
      min_t <- t_new
    }
    if(i %% 10 == 0) {
      t_history[i/10] <- t
      min_t_history[i/10] <- min_t
    }

  }

  return(list(A = min_A, t_history = t_history,
              min_t_history = min_t_history,
              iterations = MAX_ITER, theta = theta,
              PI = PI, E = E))
}


# ######################################################
#
# Simulations
#
# ######################################################
n <- c(100, 250, 500, 750, 1000, 2000, 5000, 10000)
cutsize <- matrix(0, nrow = 4, ncol = length(n))
cuttime <- matrix(0, nrow = 4, ncol = length(n))
for(i in 1:length(n)) {
  iter <- 10^5
  MAX_TIME <- 100
  W <- generate_data(n = n[i], block = FALSE)
  # ##############
  # KL
  # ##############
  if(n[i] > 2000) {
    cutsize[1, i] <- NA
  } else {
    start_time <- proc.time()
    result1 <- KL(W)
```

```
  timeA1 <- proc.time() - start_time
  while(result1$max_gain > 1e-4 & timeA1[[3]] < MAX_TIME) {
    result1 <- KL(W, result1$A)
    timeA1 <- proc.time() - start_time
  }
  A1 <- result1$A
  cutsize[1, i] <- cut_size(W, A1)
  cuttime[1, i] <- timeA1[[3]]
}

# ##############
# SA
# ##############
start_time <- proc.time()
result2 <- SA(W, initial_t = 100, MAX_ITER = iter)
timeA2 <- proc.time() - start_time
A2 <- result2$A
cutsize[2, i] <- cut_size(W, A2)
cuttime[2, i] <- timeA2[[3]]

# ##############
# SAA
# ##############
start_time <- proc.time()
result3 <- SAA(W, E = seq(result2$t_history[[1000]]*0.8,
                          result2$t_history[[1000]]*1.5,
                          length.out = 100),
               t0 = 5000, initial_t = 100, MAX_ITER = iter)
timeA3 <- proc.time() - start_time
A3 <- result3$A
cutsize[3, i] <- cut_size(W, A3)
cuttime[3, i] <- timeA3[[3]]
# ##############
# SAMC
# ##############
start_time <- proc.time()
result4 <- SAMC(W, E = seq(result2$t_history[[1000]]*0.8,
                           result2$t_history[[1000]]*1.5,
                           length.out = 100),
                t0 = 5000, MAX_ITER = iter)
timeA4 <- proc.time() - start_time
A4 <- result4$A
cutsize[4, i] <- cut_size(W, A4)
cuttime[4, i] <- timeA4[[3]]


#Create graphs for the results.
```

```
  png(paste("images/graph_min_cut_n", n[i], "_iter",
            iter, ".png", sep = ""),
      2000, 2000, res = 400)
  colors <- c("black", "orange", "blue", "gray")
  plot((1:(result2$iterations/10))*10, result2$min_t_history,
       type = "l",
       xlim = c(0, iter),
       ylim = c(min(cutsize[2:4, i]), max(result2$min_t_history)),
       ylab = "Minimum cut size", xlab = "Iteration number",
       col = colors[1], lty = 1, xaxt = "n")
  axis(1, at = 0:2*iter/2)
  lines((1:(result3$iterations/10))*10, result3$min_t_history,
        col = colors[2], lty = 2)
  lines((1:(result4$iterations/10))*10, result4$min_t_history,
        col = colors[3], lty = 3)
  legend("topright", c("SA", "SAA", "SAMC"),
         cex = 0.8, col = colors,
         lty = 1:4, lwd = 2, bty = "n")
  dev.off()

  png(paste("images/graph_cut_n", n[i], "_iter",
            iter, ".png", sep = ""),
      2000, 2000, res = 400)
  plot((1:(result2$iterations/10))*10, result2$t_history,
       type = "l",
       xlim = c(0, iter),
       ylim = c(min(cutsize[2:4, i]), max(result2$t_history)),
       ylab = "Cut size", xlab = "Iteration number",
       col = colors[1], lty = 1, xaxt = "n")
  axis(1, at = 0:2*iter/2)
  lines((1:(result3$iterations/10))*10, result3$t_history,
        col = colors[2], lty = 2)
  lines((1:(result4$iterations/10))*10, result4$t_history,
        col = colors[3], lty = 3)
  legend("topright", c("SA", "SAA", "SAMC"),
         cex = 0.8, col = colors,
         lty = 1:4, lwd = 2, bty = "n")
  dev.off()
}

#Create tables for the results.
png(paste("images/graph_all_vals_iter", iter, ".png", sep = ""),
    600, 140, res = 100)
vals <- round(cutsize)
colnames(vals) <- n
rownames(vals) <- c( "KL", "SA", "SAA", "SAMC")
grid.table(vals)
```

```
dev.off()

png(paste("images/graph_all_times_iter", iter, ".png", sep = ""),
    2000, 2000, res = 400)
plot(n[n <= 2000], cuttime[1, n <= 2000], type = "l",
     col = colors[4], lty = 4, ylim = c(0, max(cuttime)),
     ylab = "Total CPU time (s)", xlab = "n",
     xlim = c(n[1], n[length(n)]))
lines(n, cuttime[2, ], col = colors[1], lty = 1)
lines(n, cuttime[3, ], col = colors[2], lty = 2)
lines(n, cuttime[4, ], col = colors[3], lty = 3)
legend("topright", c("KL", "SA", "SAA", "SAMC"), cex = 0.8,
       col = colors[c(4, 1, 2, 3)], lty = c(4, 1, 2, 3),
       lwd = 2, bty = "n")
dev.off()
```

8 References

Gary, M. R., & Johnson, D. S. (1979). Computers and intractability: A guide to the theory of nP-completeness. WH Freeman; Company, New York.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, *37*(6), 865–892.

Karypis, G., & Kumar, V. (1995). Multilevel graph partitioning schemes. In *ICPP (3)* (pp. 113–122).

Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, *49*(2), 291–307.

Kirkpatrick, S. (1984). Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, *34*(5-6), 975–986.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simmulated annealing. *Science*, *220*(4598), 671–680.

Liang, F., Cheng, Y., & Lin, G. (2014). Simulated stochastic approximation annealing for global optimization with a square-root cooling schedule. *Journal of the American Statistical Association*, *109*(506), 847–863. https://doi.org/10.1080/01621459.2013.872993

Liang, F., Liu, C., & Carroll, R. J. (2007). Stochastic approximation in monte carlo computation. *Journal of the American Statistical Association*, *102*(477), 305–320.