The Complexity of the k-means Method*

Tim Roughgarden[1] and Joshua R. Wang[2]

1 **Department of Computer Science, Stanford Univeresity, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305, USA**
tim@cs.stanford.edu

2 **Department of Computer Science, Stanford Univeresity, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305, USA**
joshua.wang@cs.stanford.edu

Abstract

The k-means method is a widely used technique for clustering points in Euclidean space. While it is extremely fast in practice, its worst-case running time is exponential in the number of data points. We prove that the k-means method can implicitly solve PSPACE-complete problems, providing a complexity-theoretic explanation for its worst-case running time. Our result parallels recent work on the complexity of the simplex method for linear programming.

1998 ACM Subject Classification I.5.3 Clustering

Keywords and phrases k-means, PSPACE-complete

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.252

1 Introduction

The k-means method, also known as Lloyd's algorithm [15], is a widely used technique for clustering points in Euclidean space. It can be viewed as a local search algorithm for the problem of, given n data points in $\mathbb{R}^d$, choosing k centers in $\mathbb{R}^d$ to minimize the sum (or average) of squared Euclidean distances between each point and its closest center.[1] The method begins with an arbitrary set of k initial centers. Each point is then reassigned to the center closest to it, and each center is recomputed as the center of mass of its assigned points. Every iteration decreases the objective function value of the clustering, and these two steps are repeated until the algorithm stabilizes.

Three basic facts about the k-means method are:

1. It is extremely fast in practice, and for this reason is widely used, perhaps more than any other clustering algorithm. For example, Berkhin [6] states that it "is by far the most popular clustering algorithm used nowadays in scientific and industrial applications."
2. The worst-case running time of the method is exponential in the number of points. This was first proved by Arthur and Vassilvitskii [4], and extended to the plane by Vattani [18].
3. It has polynomial smoothed complexity in the sense of Spielman and Teng [17]: for every choice of data points, in expectation over Gaussian perturbations with standard deviation σ of these points, the running time of the method is polynomial in the input size and in $1/\sigma$ [3].[2]

* This work was supported in part by NSF grant CCF-1524062 and a Stanford Graduate Fellowship.

[1] This problem is NP-hard, even in the plane [16].

[2] We focus on properties that concern the running time of the k-means method. Like with any local search algorithm, one can also consider the approximation quality of the solution output by the method; see the well-known k-means++ method [5] for an initialization technique with a provable guarantee,

These three properties of the k-means method illustrate a clear parallel with the simplex method for linear programming. The simplex method is famously fast in practice, but Klee and Minty [14] showed that it has exponential worst-case running time. These lower bounds have since been extended to many different pivot rules (see e.g. Amenta and Ziegler [2]), and also to restricted classes of linear programs, such as minimum-cost flow [19]. On the other hand, both the average-case and smoothed running times of the simplex method are polynomial (see Spielman and Teng [17] and the references therein).

Disser and Skutella [9] initiated a fresh take on the worst-case exponential running time of the simplex method, by showing that it inadvertently solves problems that are much harder than linear programming. Specifically, they showed how to efficiently embed an instance of the (NP-complete) Partition problem into a linear program so that the trajectory of the simplex method immediately reveals the answer to the instance. In this sense, the simplex method can solve NP-hard problems, thereby providing an explanation of sorts for its worst-case running time. A similar line was taken by Adler et al. [1], who exhibited a pivot rule with which the simplex method can solve PSPACE-complete problems, and Fearnley and Savani [10], who proved analogous results with Dantzig's original pivot rule. These results echo earlier works on PLS-complete local search problems, where computing the specific local minimum computed by local search is a PSPACE-complete problem (assuming completeness is proved using a "tight" reduction, as in almost all known examples) [12], and the results of Goldberg et al. [11] showing that computing the outcome of various algorithms that solve $PPAD$-complete problems, such as the Lemke-Howson algorithm for computing a Nash equilibrium, are PSPACE-complete problems.

Our contribution is a proof that the k-means method, just like the simplex method, inadvertently solves PSPACE-complete problems. That is: computing the outcome of the k-means method, given an instance of k-means and an initialization for the k centers, is a PSPACE-complete problem.[3] Like with the earlier results on the simplex method, this result provides a new interpretation of the worst-case running time of the k-means method — it is exponential not because the work done is inherently wasteful, but rather because it solves a much harder problem than the one it was originally designed for. Our result also implies, under appropriate complexity assumptions, that there is no way of significantly "speeding up" the k-means method (in the worst case) without changing its final state.

▶ **Theorem 1.** *Given a k-means input $(\mathcal{X},\mathcal{C})$, it is PSPACE-hard to determine the final cluster centers.*

2 Preliminaries

We briefly review the C-PATH problem, which serves as the starting point for our reduction. The C-PATH problem was used by Adler, Papadimitriou, and Rubinstein to show that determining whether a particular basis occurs on the path of the simplex algorithm, under certain pivoting rules, is PSPACE-complete [1]. The C-PATH problem is as follows: we are given as input a boolean circuit C with fan-in 2 which takes in its own input of n bits, and a target binary string t. For every input x, $C(x)$ is at most Hamming distance one from x, i.e.

and [8, 7] for matching lower bounds on this particular method. Constant-factor guarantees are also known for different local search algorithms [13].

[3] Determining the complexity of computing *any* local minimum of the local search problem corresponding to the k-means method — not necessarily the local minimum computed by the method on a given initialization — is an intriguing open problem.

C computes an index (if any) to flip. Note that this means that C's input and output are the same size. Suppose we begin with the all-zeroes binary string and repeatedly apply C. The sequence we get, $(0, C(0), C(C(0)), \ldots)$, is the path of C. We want to compute whether this path includes t.

▶ **Lemma 2** ([1]). *There is a family of circuits C of size polynomial in the number of inputs and of polynomial complexity such that C-PATH is PSPACE-complete.*

3 Reduction Sketch

In this section, we sketch the reduction we use to prove Theorem 1. For the sake of clarity and brevity, we omit some technical details here which are addressed in the full construction.

Reminder of Theorem 1 *Given a* `k-means` *input $(\mathcal{X}, \mathcal{C})$, it is PSPACE-hard to determine the final cluster centers.*

Recall that we reduce from the C-PATH problem, where we have a circuit C and target binary string t. We know that if the path of C ever reaches t, it must do so within 2^n steps. Our plan of attack is to encode circuit C into `k-means`, and then use the reset gadget of Arthur and Vassilvitskii [4] to repeatedly run the encoded circuit on its own output. It is worth noting that although Vattani [18] showed that `k-means` can be made to run for an exponential number of iterations even in the plane, this planar construction is different in a fundamental detail that we depend on. In Arthur and Vassilvitskii's construction, a reset gadget is capable of resetting *all* earlier cluster centers. In Vattani's construciton, a reset gadget resets only the previous cluster center, but does so twice. These gadgets both suffice when the base instance is a single cluster, but only the former can handle a more complex base instance with multiple clusters.

Encoding the Circuit

One benefit of choosing the C-PATH problem is that encoding the circuit is simply a matter of encoding its gates. We use the location of a certain cluster center to represent a boolean value of our gate. When we compute that a gate evaluates to false, its cluster center moves from a starting location to a false region. If it evaluated to true, it would move into a disjoint true region instead.

We can assume without loss of generality that our circuit only uses NAND gates. We go through these gates in topological order; with each new NAND gate we introduce a new dimension to our `k-means` instance. Hence, the inputs to our current gate always lie in a lower-dimensional space. Our NAND gate has two inputs, each with their own false and true region in the space below. Suppose we place an intermediate point roughly $d - \epsilon$ units above each of these regions which are part of cluster i whose center is currently another d units above them, for some large distance $d > 0$ and tiebreaking constant $\epsilon > 0$. When the input cluster centers move to their false or true regions, they steal the respective point above them from cluster i. Depending on which points are stolen, the center of cluster i moves to a predictable location. With additional arranging of the intermediate points, the center moves to either a false region or a true region, the two of which are disjoint.

Repeatedly Running the Circuit

Unfortunately, we cannot immediately apply Arthur and Vassilvitskii's reset gadgets, because they are designed to return a set of cluster centers from specific final locations to initial

locations [4]. We care about the final locations of our cluster centers because they represent the output value of our circuit and should affect the input value for the next iteration.

We solve this by, instead of using a single AV reset gadget to reset all cluster centers, using two AV reset gadgets for each input/output bit. One gadget detects when the output bit is false and returns the cluster center to its initial position while also setting the input bit to false. The other gadget does the same but for true bits. Furthermore, we can still layer these gadgets; gadgets reset all corresponding gadgets in previous layers. With only n layers, we can run the circuit 2^n times, enough to guarantee that t will appear if it is indeed in the path of the circuit.

As a final step, we add a gadget to track whether t has appeared. To do this, suppose we modify the circuit so that a special bit is 1 if and only if the input was t. We can use the same idea as with NAND gates; we add an intermediate point roughly $d - \epsilon$ units above the true region of this special bit, which are part of a cluster whose center is an additional d units above the intermediate point. The only other point in this cluster, in fact, is d units above the center. If the special bit ever becomes 1, the intermediate point will be stolen and the cluster center will move to the top point. After this, the center is $2d$ from any other point and can neither gain nor lose data points. All we need to check in the `k-means` output is the location of this center to know whether the path of C includes t.

4 Formal Reduction Proof

In this section, we formally prove Theorem 1. We follow the sketch given in Section 3. Missing tables can be found in Appendix A.

Encoding the Circuit

Recall that we begin with an instance of the C-PATH problem, (C, t) where we want to know if t is on the path of C. It will be convenient to convert C so that it only has NAND gates instead of the standard AND, OR, and NOT gates. We also require that each gate has a fan-out of at most two, but introduce a special SPLIT gate which takes in a single bit and outputs it back. This can be implemented with only a constant blowup in the number of gates, since fan-in larger than two can be simulated with a binary tree of SPLIT gates (and the number of nodes is at most twice the number of leaves). We require that the inputs of a NAND gate can only SPLIT gates, which can be guaranteed by inserting a SPLIT gate of fan-out one before each NAND gate. This at most doubles the number of gates. We also require that the inputs of a NAND gate must be at the same depth and that all outputs are at the same depth. One slightly inefficient, yet still polynomial method to guarantee this is to take every NAND gate and place it in its own layer. Inside a layer, there is only a single NAND gate, but we use SPLIT gates to pass on the other values. We add SPLIT gates after outputs which occur too early. This synchronizes the circuit with only a quadratic blowup in the number of gates.

We represent boolean values in our circuit with the location of a cluster center. Each cluster center serves to signal the output of a gate to only one other gate (why we bound the fan-out). Gate i uses cluster centers c_{2i-1} and c_{2i}. Cluster center c_j has an initial location $\mathbf{s}_j$, a false region centered at $\mathbf{b}_{j,0}$ with radius r_j, a true region centered at $\mathbf{b}_{j,1}$ with the same radius r_j, and a final location $\mathbf{t}_j$. At some timestep, the center will move from its initial location to either its false region or its true region, which serves as a signal to the gate that takes it as input. It then eventually moves to its final location. We guarantee that no two initial locations, false regions, true regions, or final locations overlap, even over different

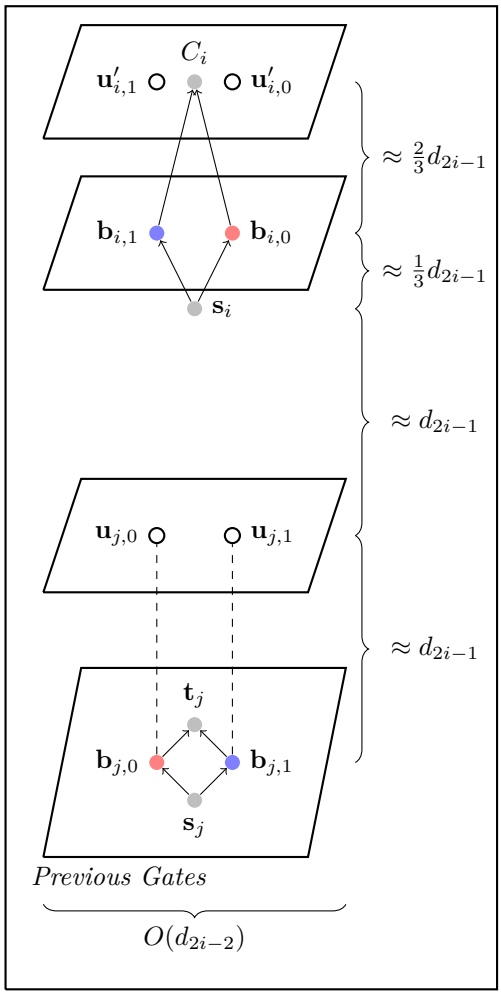

Figure 1 SPLIT Gadget, Upper Half

clusters. This property remains true when moving the final location towards the average of the false and true region centers. Finally, we guarantee a cluster center is always the closest center to any of its locations or regions.

We construct gates in topological order. For each gate i, we introduce two new dimensions: $(2i-1)$ and $(2i)$. We let $\mathbf{e}_j$ be the standard basis vector for dimension j. We also grow the scale of our construction at each step; for each dimension, we choose a scaling factor $d_i > 0$ so that $d_1 \ll d_2 \ll \cdots \ll d_{2m}$. The data points we introduce with dimension i are within $O(d_i)$ of the origin. The idea is that d_{i+1} is large enough compared to d_i so that two points which differ by d_{i+1} in their $(i+1)^{th}$ coordinate and by $O(d_i)$ in their first i coordinates are still roughly d_{i+1} apart. We also use a small $\epsilon > 0$ to break ties (note $\epsilon \ll d_1$). We also use $d(\mathbf{u}, \mathbf{v})$ to represent the Euclidean distance between points $\mathbf{u}$ and $\mathbf{v}$.

SPLIT Gadget

We first explain the construction of the simpler SPLIT gadget. Suppose we have the i^{th} gate which takes the j^{th} cluster center as input. The data points and regions we use in this construction are listed in Table 1 and the upper half of the gadget is depicted in Figure 1.

Suppose that at time T, the j^{th} cluster center moves to its false or true region. We want to notice when this occurs, so we place an intermediate point $\mathbf{u}_{j,0}$ roughly $(d_{2i-1} - \epsilon)$ above the false region, an intermediate point $\mathbf{u}_{j,1}$ roughly $(d_{2i-1} - \epsilon)$ above the true region, an intermediate point $\mathbf{v}_{j,0}$ roughly $(d_{2i-1} - \epsilon)$ below the false region, and an intermediate point $\mathbf{v}_{j,1}$ roughly $(d_{2i-1} - \epsilon)$ below the true region. Note that the actual heights are actually scaled to account for the radius of each of these regions; *every* point in the region at most $(d_{2i-1} - \epsilon)$ away from its two intermediate points.

We want the $(2i-1)^{th}$ cluster center to be d units away from the top two intermediate points and the $(2i)^{th}$ cluster center to be d units away from the bottom two intermediate points. For each intermediate point u, we add a counterbalancing point u' so that the average of the two is our desired initial center location.

At time $T+1$, the j^{th} cluster steals either $\mathbf{u}_{j,0}$ and $\mathbf{v}_{j,0}$ or $\mathbf{u}_{j,1}$ and $\mathbf{v}_{j,1}$, depending on whether it was false or true. This causes the $(2i-1)^{th}/(2i)^{th}$ cluster centers to move up/down to their respective false or true regions (which are actually balls of radius zero). This does not affect the $(2i-1)$-coordinate of the j^{th} cluster since the two points it stole cancel out. However, it does move the center towards the center of the region it was in.

At time $T+2$, because the $(2i-1)^{th}$ and $(2i)^{th}$ cluster centers moved away from the lower-dimensional space, the other intermediate points are stolen by cluster j (recall we guarantee that cluster center j is the closest center to any of its regions, and in particular to the false or true region it was not in). This causes the $(2i-1)^{th}/(2i)^{th}$ cluster centers to move up/down to their final locations. Again, this does not affect the $(2i-1)$-coordinate of the j^{th} cluster because the points cancel out. However, it does move the center towards the center of the region it was not in. Notice we have affected the final location of the j^{th} cluster center, but we already assumed that moving it towards the average of the false and true regions would keep all locations and regions disjoint.

We see that we satisfy the assumptions made about the construction; the locations and regions we create are disjoint from all others because the other locations and regions are within $O(d_{2i-2})$ of the origin and all of ours are at least $\Omega(d_{2i-1})$ from the origin. This also makes our cluster center the closest to all of our locations and regions (since other centers cannot escape the lower-dimensional space due to our balancing). Finally, moving our final location towards the average of the false and true regions keeps it disjoint.

NAND Gadget

We now proceed to the constrution of the NAND gadget. Suppose we have the i^{th} gate which takes the j^{th} and k^{th} cluster centers as input. For simplicity, we concern ourselves with the data points and cluster center regions for cluster $(2i-1)$ only. The $(2i)^{th}$ cluster's points and regions can be found by negating the $(2i-1)^{th}$ and $(2i)^{th}$ coordinates. The points and regions we do present are listed in Table 2 and depicted in Figure 2. For this gadget, we talk about dimension $2i-1$ as left/right and dimension $2i$ as up/down.

We have the same core plan as the SPLIT gadget. We add intermediate points above the two false regions and above the two true regions. We also slide these intermediate points left and right slightly; we shift intermediate points above false regions to the right and intermediate points above true regions to the left. Because $d_{2i-1} \ll d_{2i}$, these points are still roughly $d_{2i} - \epsilon$ from their respective regions.

The initial center of cluster $(2i-1)$ is d_{2i} above all four intermediate points; we achieve this by adding counterbalancing points so that the average of a point and its counterbalancing partner is our desired center.

Suppose at time T both cluster centers j and k move to false or true regions. Then at

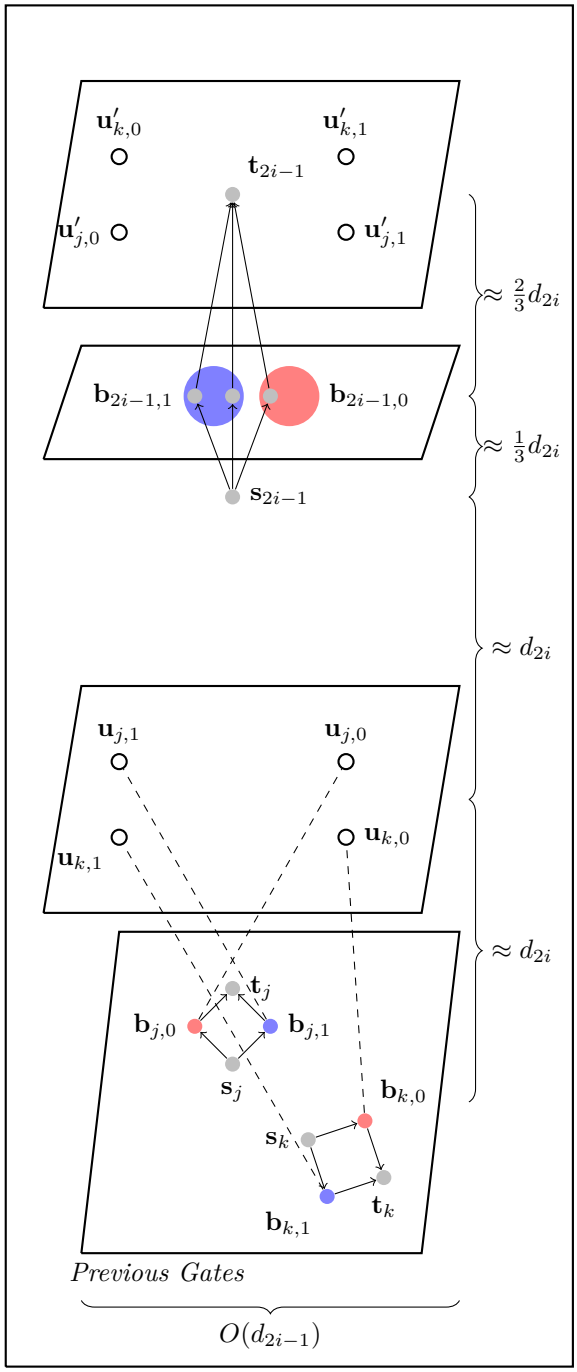

Figure 2 NAND Gadget, Upper Half

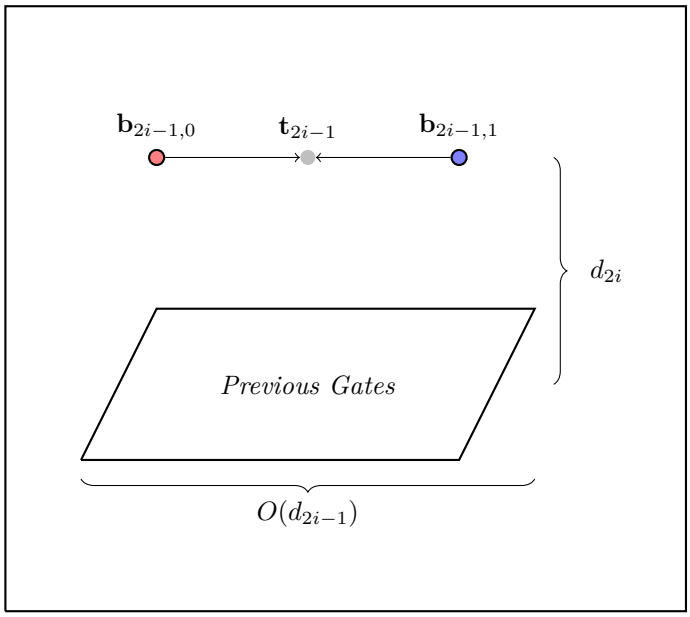

Figure 3 Input Gadget

time $T+1$, they each steal an appropriate intermediate point above them. This causes the center of cluster $(2i-1)$ to definitely upwards, and possibly left or right depending on what values j and k had. If both j and k are false, they steal intermediate points on the right and our center moves to the left. If they are both true, our center moves to the right. If we have one of each, the center does not move left or right. We can place our false region to capture the right possibility and our true region to capture both the center and left possibilities.

At time $T+2$, cluster centers j and k steal the intermediate point for the region they did not enter, because cluster center $(2i-1)$ moved up and they are the closest center to any of their respective regions. This moves the cluster center $(2i-1)$ to its final location. Note that we have shifted the final location of cluster centers j and k, but this keeps locations and regions disjoint by assumption.

We satisfy all assumptions made about the construction for essentially the same reasons as before. The one difference is regarding moving our final location towards the average of the false and true regions. It remains disjoint because $d_{2i-1} \ll d_{2i}$, so it descends approximately straight from above, avoiding the false and true regions.

Input Gadget

We also provide a simple gadget which is used to signal the value of an input bit. We only use the $(2i-1)^{th}$ cluster center and do not actually provide an initial location. Instead, the center is intended to start in either the false or true region, which signals its value. It then immediately moves to its final location. Dimension $(2i-1)$ is used to separate the false and true regions while dimension $(2i)$ is used to separate the gadget from previous gadgets. The points and regions are listed in Table 3 and depicted in Figure 3. Note that unlike previous gadgets, there is no corresponding bottom half to this gadget, since we do not need to balance its effect on previous gadgets.

AV Reset Gadget Review

Before we describe how we repeatedly run the circuit, we briefly review the reset gadgets of Arthur and Vassilvitskii [4]. A configuration is signaling if at least one final cluster center is distinct from every cluster center arising in previous iterations. A configuration is super-signaling if all final cluster centers are distinct from every cluster center arising in previous iterations and there is an alternate initial configuration of centers that is essentially identical except at least one final cluster center is different.

AV gadgets are a result of two constructions (Lemma 3.3 and Lemma 3.4 in their paper). One construction (Lemma 3.3) converts a super-signaling configuration into a signaling configuration which runs from the initial configuration and then swaps to the alternate intial configuration. The other (Lemma 3.4) converts a signaling configuration into a super-signaling configuration. They share similar ideas to our gadget constructions above; intermediate points are placed above and below the expected locations. Similar to our NAND gadget shifting the intermediate points left/right, they shift the intermediate points in a *circle* using two additional dimensions. They then use a second set of intermediate points which are stolen according to how points were shifted in a circle. This enables them to correct every center to its new initial location. There are also two additional clusters which represent an alternate initial configuration.

Taken together, these two constructions take a lower-dimensional signaling configuration and make it run twice by resetting its final cluster positions to their initial positions. Layering n gadgets results in 2^n resets of the bottom-level circuit.

We make two key observations about the capabilities of AV reset gadgets. First, they need not reset the positions of all lower-dimensional clusters; we be selective and only reset some. Second, the signaling cluster may reach its distinct location one step before it reaches its final position. This works because intermediate points may be placed above and below its distinct location, and the signaling cluster will steal them and reach its final position in the same time step. The location of the intermediate point which corrects its final location to its initial location still uses its real final location. These observations enable us to use AV reset gadgets to set the input of the circuit to its previous output, despite our particular method of signalling boolean values.

Repeatedly Running the Circuit

We are now ready to explain how to repeatedly run our encoded circuit. We need to run it up to 2^n times to guarantee we reach t, if at all possible. We plan to do this with AV gadgets. Unfortunately, AV reset gadgets work by knowing the exact final locations of cluster centers and moving them to exact initial locations. We want to copy circuit output to input. Not only do we not have exact final locations, but we also want them to influence the new initial locations. This is solved by using one AV reset gadget chain per input bit and each boolean value it can take on. An AV reset gadget for bit i being false is signaled by the cluster center of output bit i entering its false region. It corrects the final location of the cluster center to the false region of input bit i and it corrects the final location of input bit i to the initial location of output bit i.

We want to run the circuit 2^n times, so a first attempt is to reset each (input bit, value) pair that many times. Unfortunately, this has unintended behavior if the path of C has unbalanced parity. For example, suppose we had a circuit C where $C(00) = 01$ and $C(01) = 00$. In four iterations, we follow the path $(00, 01, 00, 01, 00)$. The false AV reset gadget for the first input bit is now fully expended and stops resetting, but the AV reset

gadgets for the second input bit still reset it, causing only part of the circuit input to be copied from the output. In a more complex example, this could evaluate the circuit on an input not actually in the path of C.

To avoid this, we transform C so that it follows a balanced path, i.e. in every bit it alternates between true and false. Circuit C_2 has an additional parity bit in its input. When this auxiliary bit is zero, it simply flips the entire input. When the auxiliary bit is one, it again flips the entire input, and then applies C_2 to the standard input. This can be be implemented with only polynomially many extra gates. More formally:

$$C_2(\mathbf{x}b_1) = \begin{cases} \bar{\mathbf{x}}1 & \text{if } b_1 = 0 \\ C(\bar{\mathbf{x}})0 & \text{if } b_1 = 1 \end{cases}$$

We now have $n+1$ input bits and plan to reset each 2^n times, for each value. It will also be convenient to keep track of whether we have reached t yet, so that we can simply examine the final `k-means` state. We add another auxiliary bit, which transitions from zero to one when the input is t. This also can be implemented with only polynomially many extra gates. Formally:

$$C_3(\mathbf{x}b_1b_2) = \begin{cases} C_2(\mathbf{x}b_1)0 & \text{if } b_2 = 0, \mathbf{x}b_1 \neq t0 \\ C_2(\mathbf{x}b_1)1 & \text{if } b_2 = 0, \mathbf{x}b_1 = t0 \\ C_2(\mathbf{x}b_1)1 & \text{if } b_2 = 1 \end{cases}$$

Note that for our second auxiliary bit, we want to reset a false value 2^{n+1} times and we do not worry about resetting a true value. Also, we can conveniently use the AV reset widget of this bit to reset the inner gates of the circuit. This completes our circuit reset construction.

Output Gadget

Our final gadget records whether $b_2 = 1$ at any point. Suppose that b_2 is represented by the position of cluster center i. We assume without loss of generality that it is computed by a SPLIT gate. We use one final additional dimension, with the largest scale. Suppose this is dimension D. We add points at $(\mathbf{b}_{i,1} + (d_D - \epsilon)\mathbf{e}_D)$ and $(\mathbf{b}_{i,1} + (3d_D - \epsilon)\mathbf{e}_D)$. We also add a final cluster center at $(\mathbf{b}_{i,1} + (2d_D - \epsilon)\mathbf{e}_D)$. The former point can only be stolen from this cluster if $b_2 = 1$, and when this happens the cluster center will move to the latter point. But the latter point is $2d_D$ from the former point, so the cluster center can never recapture it.

As review, the completed construction uses gadgets in the following order (from low-dimensional to high-dimensional):

1. $n+2$ input gadgets which represent the input bits,
2. $poly(n)$ NAND and SPLIT gadgets which represent gates and output bits,
3. $(n+1)(2n) + (n+1)$ reset gadgets to repeatedly run the circuit,
4. and one output gadget which represents the final result.

We have produced a polynomially-sized `k-means` instance from a C-PATH instance. The final state of our output gadget indicates the answer to the C-PATH instance, so computing the final state of `k-means` is enough to solve a PSPACE-complete problem. This completes the proof.

5 Conclusions

This paper proved that the `k-means` method inadvertently solves PSPACE-complete problems, echoing analogous results for the simplex method [1, 9, 10]. There are at least three interesting directions in which our result might be extended.

1. We conjecture that the following problem is *PLS*-complete: given an instance of `k-means`, compute an arbitrary local minimum of the `k-means` method. Such a result, if proved using "tight" reductions[4] (see [12]), would imply our Theorem 1.[5]
2. The worst-case running time of the `k-means` method is exponential even in two dimensions [18], while our PSPACE-completeness reduction produces instances with a large number of dimensions. Is computing the outcome of `k-means` still PSPACE-complete in planar instances? Recall that Vattani's reset gadgets work by resetting only the previous gadget, but doing so twice. Our reduction depended on the ability of AV reset gadgets to reset all previous clusters so that the entire circuit could be reset.
3. Does the problem of computing the outcome of `k-means` remain PSPACE-complete when the initial centers are chosen greedily, as in `k-means`++ [5]?

References

1 Ilan Adler, Christos Papadimitriou, and Aviad Rubinstein. On simplex pivoting rules and complexity theory. In *Integer Programming and Combinatorial Optimization*, pages 13–24. Springer, 2014.

2 Nina Amenta and Gunter M Ziegler. Deformed products and maximal shadows of polytopes. *Contemporary Mathematics*, 223:57–90, 1999.

3 David Arthur, Bodo Manthey, and H Roglin. k-means has polynomial smoothed complexity. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 405–414. IEEE, 2009.

4 David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153. ACM, 2006.

5 David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

6 Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

7 Anup Bhattacharya, Ragesh Jaiswal, and Nir Ailon. A tight lower bound instance for k-means++ in constant dimension. In *Theory and Applications of Models of Computation*, pages 7–22. Springer, 2014.

8 Tobias Brunsch and Heiko Röglin. A bad instance for k-means++. In *Theory and Applications of Models of Computation*, pages 344–352. Springer, 2011.

9 Yann Disser and Martin Skutella. The simplex algorithm is np-mighty. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 858–872. SIAM, 2015.

[4] In a tight PLS reduction, there is also a correspondence between improving moves in the two instances, not just between solutions.

[5] For this question to be interesting, it is important to rule out degenerate local minima. A partial solution is to insist that each of the k initial centers lies in the convex hull of the point set. (Otherwise, placing one center at the center of mass of the entire point set and the other $k-1$ centers "at infinity" is an easy-to-compute local minimum.) Similarly, dealing with (or avoiding) co-located centers requires some care.

10 John Fearnley and Rahul Savani. The complexity of the simplex method. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 201–208. ACM, 2015.

11 Paul W Goldberg, Christos H Papadimitriou, and Rahul Savani. The complexity of the homotopy method, equilibrium selection, and lemke-howson solutions. *ACM Transactions on Economics and Computation*, 1(2):9, 2013.

12 David S Johnson, Christos H Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of computer and system sciences*, 37(1):79–100, 1988.

13 Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 10–18. ACM, 2002.

14 Victor Klee and George J Minty. How good is the simplex algorithm. Technical report, DTIC Document, 1970.

15 Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

16 Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. In *WALCOM: Algorithms and Computation*, pages 274–285. Springer, 2009.

17 Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

18 Andrea Vattani. K-means requires exponentially many iterations even in the plane. *Discrete & Computational Geometry*, 45(4):596–616, 2011.

19 Norman Zadeh. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming*, 5(1):255–266, 1973.

A Missing Reduction Tables

Data Point	Location	Purpose
$\mathbf{u}_{j,0}$	$\mathbf{b}_{j,0} + \left(\sqrt{(d_{2i-1} - \epsilon)^2 - r_j^2}\right) \mathbf{e}_{2i-1}$	Detect j is False
$\mathbf{u}_{j,1}$	$\mathbf{b}_{j,1} + \left(\sqrt{(d_{2i-1} - \epsilon)^2 - r_j^2}\right) \mathbf{e}_{2i-1}$	Detect j is True
$\mathbf{u}'_{j,0}$	$\mathbf{u}_{j,1} + \left(\sqrt{4d_{2i-1}^2 - d(\mathbf{u}_{j,0}, \mathbf{u}_{j,1})}\right) \mathbf{e}_{2i-1}$	Counterbalance $\mathbf{u}_{j,0}$
$\mathbf{u}'_{j,1}$	$\mathbf{u}_{j,0} + \left(\sqrt{4d_{2i-1}^2 - d(\mathbf{u}_{j,0}, \mathbf{u}_{j,1})}\right) \mathbf{e}_{2i-1}$	Counterbalance $\mathbf{u}_{j,1}$
$\mathbf{v}_{j,0}$	$\mathbf{b}_{j,0} - \left(\sqrt{(d_{2i-1} - \epsilon)^2 - r_j^2}\right) \mathbf{e}_{2i-1}$	Detect j is False
$\mathbf{v}_{j,1}$	$\mathbf{b}_{j,1} - \left(\sqrt{(d_{2i-1} - \epsilon)^2 - r_j^2}\right) \mathbf{e}_{2i-1}$	Detect j is True
$\mathbf{v}'_{j,0}$	$\mathbf{u}_{j,1} - \left(\sqrt{4d_{2i-1}^2 - d(\mathbf{u}_{j,0}, \mathbf{u}_{j,1})}\right) \mathbf{e}_{2i-1}$	Counterbalance $\mathbf{v}_{j,0}$
$\mathbf{v}'_{j,1}$	$\mathbf{u}_{j,0} - \left(\sqrt{4d_{2i-1}^2 - d(\mathbf{u}_{j,0}, \mathbf{u}_{j,1})}\right) \mathbf{e}_{2i-1}$	Counterbalance $\mathbf{v}_{j,1}$
Cluster Center Region	**Center**	**Radius**
$\mathbf{s}_{2i-1}$	$\frac{1}{4}\left(\mathbf{u}_{2i-1,0} + \mathbf{u}_{2i-1,1} + \mathbf{u}'_{j,0} + \mathbf{u}'_{j,1}\right)$	0
$\mathbf{b}_{2i-1,0}$	$\frac{1}{3}\left(\mathbf{u}_{2i-1,1} + \mathbf{u}'_{j,0} + \mathbf{u}'_{j,1}\right)$	0
$\mathbf{b}_{2i-1,1}$	$\frac{1}{3}\left(\mathbf{u}_{2i-1,0} + \mathbf{u}'_{j,0} + \mathbf{u}'_{j,1}\right)$	0
$\mathbf{t}_{2i-1}$	$\frac{1}{2}\left(\mathbf{u}'_{j,0} + \mathbf{u}'_{j,1}\right)$	0
$\mathbf{s}_{2i}$	$\frac{1}{4}\left(\mathbf{v}_{2i-1,0} + \mathbf{v}_{2i-1,1} + \mathbf{v}'_{j,0} + \mathbf{v}'_{j,1}\right)$	0
$\mathbf{b}_{2i,0}$	$\frac{1}{3}\left(\mathbf{v}_{2i-1,1} + \mathbf{v}'_{j,0} + \mathbf{v}'_{j,1}\right)$	0
$\mathbf{b}_{2i,1}$	$\frac{1}{3}\left(\mathbf{v}_{2i-1,0} + \mathbf{v}'_{j,0} + \mathbf{v}'_{j,1}\right)$	0
$\mathbf{t}_{2i}$	$\frac{1}{2}\left(\mathbf{v}'_{j,0} + \mathbf{v}'_{j,1}\right)$	0

Table 1 SPLIT Gadget Points and Regions

Data Point	Location	Purpose
$\mathbf{u}_{j,0}$	$\mathbf{b}_{j,0} + d_{2i-1}\mathbf{e}_{2i-1} + (d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Detect j is False
$\mathbf{u}_{j,1}$	$\mathbf{b}_{j,1} - d_{2i-1}\mathbf{e}_{2i-1} + (d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Detect j is True
$\mathbf{u}_{k,0}$	$\mathbf{b}_{k,0} + d_{2i-1}\mathbf{e}_{2i-1} + (d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Detect k is False
$\mathbf{u}_{k,1}$	$\mathbf{b}_{k,1} - d_{2i-1}\mathbf{e}_{2i-1} + (d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Detect k is True
$\mathbf{u}'_{j,0}$	$-\mathbf{b}_{j,0} - d_{2i-1}\mathbf{e}_{2i-1} + (3d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Counterbalance $\mathbf{u}_{j,0}$
$\mathbf{u}'_{j,1}$	$-\mathbf{b}_{j,1} + d_{2i-1}\mathbf{e}_{2i-1} + (3d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Counterbalance $\mathbf{u}_{j,1}$
$\mathbf{u}'_{k,0}$	$-\mathbf{b}_{k,0} - d_{2i-1}\mathbf{e}_{2i-1} + (3d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Counterbalance $\mathbf{u}_{k,0}$
$\mathbf{u}'_{k,1}$	$-\mathbf{b}_{k,1} + d_{2i-1}\mathbf{e}_{2i-1} + (3d_{2i} - \epsilon)\,\mathbf{e}_{2i}$	Counterbalance $\mathbf{u}_{k,1}$
Cluster Center Region	**Center**	**Radius**
$\mathbf{s}_{2i-1}$	$(2d_{2i} - \epsilon)\mathbf{e}_{2i}$	0
$\mathbf{b}_{2i-1,0}$	$-\frac{1}{6}(\mathbf{b}_{j,1} + \mathbf{b}_{k,1}) + \frac{1}{2}d_{2i-1}\mathbf{e}_{2i-1} + (\frac{7}{3}d_{2i} - \epsilon)\mathbf{e}_{2i}$	$\frac{1}{5}d_{2i-1}$
$\mathbf{b}_{2i-1,1}$	$-\frac{1}{6}(\mathbf{b}_{j,0} + \mathbf{b}_{k,0}) - \frac{1}{6}d_{2i-1}\mathbf{e}_{2i-1} + (\frac{7}{3}d_{2i} - \epsilon)\mathbf{e}_{2i}$	$\frac{1}{5}d_{2i-1}$
$\mathbf{t}_{2i-1}$	$-\frac{1}{4}(\mathbf{b}_{j,0} + \mathbf{b}_{j,1} + \mathbf{b}_{k,0} + \mathbf{b}_{k,1}) + (3d_{2i} - \epsilon)\mathbf{e}_{2i}$	0

Table 2 NAND Gadget Points and Regions, Upper Half

Data Point	Location	Purpose
$\mathbf{v}_{2i-1,0}$	$-d_{2i-1}\mathbf{e}_{2i-1} + d_{2i}\mathbf{e}_{2i}$	False Point
$\mathbf{v}_{2i-1,1}$	$d_{2i-1}\mathbf{e}_{2i-1} + d_{2i}\mathbf{e}_{2i}$	True Point
Cluster Center Region	**Center**	**Radius**
$\mathbf{b}_{2i-1,0}$	$-d_{2i-1}\mathbf{e}_{2i-1} + d_{2i}\mathbf{e}_{2i}$	0
$\mathbf{b}_{2i-1,1}$	$d_{2i-1}\mathbf{e}_{2i-1} + d_{2i}\mathbf{e}_{2i}$	0
$\mathbf{t}_{2i-1}$	$d_{2i}\mathbf{e}_{2i}$	0

Table 3 Input Gadget Points and Regions