The Suitability of Kinesthetic Learning Activities for Teaching Distributed Algorithms

Paolo A. G. Sivilotti
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
paolo@cse.ohio-state.edu

Scott M. Pike
Computer Science
Texas A&M University
College Station, TX 77843-3112
pike@cs.tamu.edu

ABSTRACT

Kinesthetic learning is a process in which students learn by actively carrying out physical activities rather than by passively listening to lectures. Pedagogical research indicates that kinesthetic learning is a fundamental, powerful, and ubiquitous learning style. To date, efforts to incorporate this learning style within the computer science curriculum have focussed on introductory courses. Material in upper-level courses, however, can also benefit from a similar approach. In particular, courses on distributed computing, by the very nature of the material they cover, are uniquely suited to exploiting this learning technique. We have developed and piloted a collection of kinesthetic activities for a senior undergraduate or graduate-level course on distributed systems. We give detailed descriptions of these exercises and discuss factors that contribute to their success.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, algorithms

Keywords

Pedagogy, active learning, concurrency, reasoning

1. INTRODUCTION

A kinesthetic learning activity (KLA) is a pedagogical tool involving physical movement by students. As part of such an activity, students might stand, walk, talk, point, or even work with props. The key characteristics of a KLA are that (i) students are actively, physically engaged in the exposition and assimilation of classroom material, and (ii) this engagement directly supports a specific learning objective.

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

Most college courses rely on the traditional lecture-based format for instruction. Even when supplemented with visual slides, this format is primarily a passive form of education. As such, this format often suffers from decreased student engagement, frequent student inattention, and the exclusion of nonverbal learning modalities [4].

KLAs serve to offset these shortcomings. They can be used in the middle of a long lecture to re-energize the class by creating a new perspective from which to consider the topic. Beyond the short-term effect, including KLAs on a regular basis can have a fundamental impact on the classroom culture of interaction. As a side effect of these activities, students learn each other's names and become more comfortable asking questions and participating in group discussions. This raises the level of engagement during the periods of traditional lecturing. Finally, while traditional lectures appeal primarily to a single learning style, research in pedagogy indicates that multiple modalities are more effective [10], so incorporating KLAs broadens the scope of students who achieve positive learning outcomes.

Recently, kinesthetic learning has enjoyed increased prominence within the computer science education community [5, 2], including an annual workshop at SIGCSE. Many KLAs for computer science topics have been developed, most of which target the introductory and lower-level courses. Two templates have emerged as common themes in the design of KLAs for computer science topics: algorithms enacted by people and data structures built from people. Both of these templates naturally promote an awareness of concurrency (since participants can be simultaneously active) and locality (since cognitive and physical constraints limit how much a single participant can do). For this reason, KLAs (though relevant for all topics in the CS curriculum) are *particularly* well-suited to a course on distributed computing, where concurrency and locality are ubiquitous and fundamental themes.

In this paper, we present a collection of KLAs we have designed and piloted for a senior undergraduate or graduate-level course on distributed systems. Based on our experiences, we give detailed directions for conducting each activity including materials and logistics. We describe the learning objectives each activity is meant to support, and delineate the most important elements contributing to its success. We also give some general advice about how to design new KLAs. Our hope is that others will adopt these particular activities in their own distributed systems courses as well as use the examples in this collection as a pattern for developing new activities.

2. BACKGROUND

2.1 Personality Types and Learning Styles

The modern approach to psychological typology traces its origins to the work of C. G. Jung, who introduced a classic taxonomy of personality [9]. This work is the foundation for the Myers-Briggs Type Indicator, which divides personality types along four dimensions: introvert-extrovert, sensing-intuitive, thinking-feeling, and perceiving-judging [11]. This popular instrument is used in many settings including career counseling, team formation, and conflict resolution.

Personality type theory has also been applied in the educational setting, where evidence suggests the existence of a variety of learning styles (and complementary teaching styles) [1, 3]. No single learning style fits everyone. While some students assimilate visual information best, others prefer auditory information. While some students prefer information to be structured as facts about things, others prefer a structure based on relationships among things. While some students prefer starting with first principles and using deductive reasoning, others prefer starting with examples and using inductive or abductive approaches.

In order to accommodate this variety of learning styles, educators should strive for a balance of teaching styles. Incorporating kinesthetic teaching in the classroom is a step toward achieving this balance.

2.2 The Perils of Anthropomorphism

Dijkstra famously warned of the dangers of anthropomorphism in science in general and in computer science in particular [7].

One concern is that anthropomorphisms may encourage students to view algorithms operationally, rather than assertionally. Students should be encouraged to understand concurrent algorithms in terms of invariants, metrics, progress properties, etc., rather than in terms of sequences of actions and interleavings of events. A second concern is that anthropomorphic metaphors can be so compelling that students may make incorrect inferences based on the metaphor.

Computer science KLAs frequently put students in the roles of processors or data. Therefore, concerns about anthropomorphisms can easily be exacerbated through the careless use of such activities. However, kinesthetic learning does not, in itself, inherently carry these shortcomings. If properly designed and judiciously applied, KLAs can support assertional, not operational, reasoning. The powerful metaphors they promote can serve as mnemonic hooks for the most important concepts. The designer of a KLA is responsible for ensuring that it is a constructive aid supporting a desired learning objective rather than a harmful distraction. The activities described in this paper are exemplars of this approach.

3. SAMPLE KINESTHETIC ACTIVITIES

3.1 Nondeterministic Sorting

The Algorithm. For a given array of integers, the following action is repeated: A pair of elements is chosen, compared, and swapped if they are not in order [6]. The sequence in which pairs are chosen is not specified by the algorithm, but weak fairness requires that every pair be chosen infinitely often.

Description of the activity. A group of students is chosen to represent the data array. Each student represents a single element in the array and is given a sign indicating their position in the array. The sign should be easily visible, so a sheet of paper with an attached loop of string and worn around the neck works well. In addition, each student is given a small index card on which they write a random number. While the sign with the position should be easily visible to everyone, the value on the index card should be private. A student keeps the same array position (sign) for the entire activity but will swap data values (index cards).

During the activity, students mill around and arbitrarily select other students with which to compare data values. If the values on their respective index cards are out of order with respect to the students' positions in the array, the index cards are exchanged. At the beginning of the activity, students are instructed to raise their hands when they believe the array is sorted. When all hands have been raised, the students line up in order of array position and then read out the data values on their index cards to confirm that the array has been sorted.

An optional element of the activity is to use the rest of the class as "comparison processors". In this variant, students representing array elements are not allowed to see other students' index cards, even during a comparison. Instead, the pair of students go to a comparison processor and hand over their respective index cards. Only in the case of a swap do they discover the data value held by the other processor. This variant can lead into an interesting discussion of termination detection, but generally takes much longer to complete (even with many comparison processors).

Learning Objective 1. A deterministic, sequential algorithm is often an over specification.

Students can easily see that there are many possible execution sequences for this nondeterministic sorting algorithm. Some of these sequences correspond to well-known sequential algorithms, such as bubblesort or mergesort. In fact, every deterministic, in-place, comparison-based sorting algorithm corresponds to some execution sequence for the nondeterministic sorting program. Indeed, some students will naturally attempt a bubble-like execution, comparing their own data value with each array position in turn.

Of course, the important claim is the converse: Every execution sequence of the nondeterministic algorithm yields a sorted array. Students can be guided through the careful assertional proof of this claim based on the proper invariant (that the array is a permutation of the original) and metric.

Learning Objective 2. A good metric is not always obvious.

Participants have an intuitive sense of convergence towards the correct data value, but often have a difficult time making that intuition precise. Some metrics students have proposed include: (i) the number of elements in their correct (final) position, (ii) the sum of distances that data values are from their correct (final) position, and (iii) the length of the longest sorted prefix. The first two are simply not true, and the last is too coarse-grained to be useful (i.e., lots of good work can happen without affecting the length of the longest sorted prefix).

One observation that participants can easily make is that it becomes increasingly unlikely that they will swap index cards as the algorithm proceeds. Initially, about half of the comparisons result in swaps, but that fraction gradually de-

creases to zero. This observation leads directly to the formation of a correct metric: the number of out-of-order pairs. This number is monotonically non-increasing, bounded below, and decreased by every swap.

Learning Objective 3. An action system has terminated when it reaches a fixed point.

The execution of an action system consists of an infinite sequence of (nondeterministicly chosen) actions. The program is defined to have terminated when it reaches a fixed point: In the case of this sorting algorithm, no further swaps of data values can occur. It is generally not, however, immediately obvious to any one participant that this condition holds! This difficulty of locally recognizing the fixed point is reflected in the fact that participants hesitate in raising their hands to indicate they believe the algorithm has terminated.

To test for termination, participants tend to methodically compare with each element in the array, keeping track of each value. After the activity, they can be asked to precisely characterize the condition under which they can raise their hands. This question is even more interesting under the variant where a comparison processor is used, so participants do not see the other data value when no swap occurs. Again, the process of formalizing their intuition into a more rigorous statement is a useful one. For example, students can be asked to prove or disprove the following (erroneous) claim: A participant has their final value if they have compared with all other participants and found they did not need to swap. Post-activity questions of this kind can be used to highlight the nature of rigorous assertional reasoning.

Tips for success. Participants should not be able to easily guide the algorithm to completion by intelligently selecting comparisons. To this end, data values should be kept small (e.g., placed on index cards as described above) and private. If participants can see data values from a distance, they will tend to perform many implicit comparisons as they mill around looking for someone with whom to swap values. This mode of making many implicit comparisons creates the false impression that relatively few comparisons are needed to complete execution of the algorithm and that almost all comparisons are effective comparisons, i.e., resulting in a swap of data values. Uniqueness of data values is not necessary. In fact, the presence of duplicate values will reduce the time needed for the algorithm to complete.

It may be tempting to increase participation by running the activity with a larg array size. However, when the array is too large, the activity takes too long to complete. Conversely, if the array is too small, the amount of interesting work to be done in sorting is too small. In our experience, an array size around 10 (and no more than 15) works best.

3.2 Parallel Garbage Collection

The Algorithm. For a directed graph with a single distinguished vertex, called the *root*, vertices reachable from the root are termed *food*. All other vertices are termed *garbage*. The task is to distinguish food and garbage so that the latter can be collected. The challenge lies in accomplishing this task concurrently with a mutator process that is allowed to modify the edges (but not the set of vertices). The mutator process is allowed to (i) delete any edge and (ii) add any edge so long as the new edge is directed toward a vertex that is already food.

The trivial algorithm of marking all vertices is not correct because it marks vertices that are initially garbage. The obvious algorithm of marking the root and then propagate marks to any neighbor of a marked node is also not correct, because the mutation of the graph could undermine the diffusion of the marks. Some food may never get marked. The correct algorithm extends the mark propagating approach by also marking a node when an edge is created that is directed towards it [6].

Description of the activity. A group of 15 to 20 students is chosen to represent the vertices in the graph. Each student is given a hat and a stack of approximately 12 index cards. One student is distinguished as the root. Each student then writes their own name, once on each index card in their stack. Three index cards are collected from each student, shuffled, then redistributed to the group. The redistributed cards received by a student are *edge cards* and should be held in one's hand while the cards that were never collected are *reserves* and can be kept in one's pocket. An edge card represents an edge directed from the student holding the card to the student whose name is on the card.

Graph mutation is accomplished by the students themselves. Edges are always added and deleted by the *source* vertex. A student can delete an edge, at any time, by simply discarding an edge card from their own hand. (These discards should also be brought back periodically to the students whose names appear on the cards, just to replenish students' reserves.) To add an edge, a student must first choose a destination vertex that is reachable from the root. This is done by examing the edge cards of the root and choosing one vertex, then examing the edge cards held by that vertex and again choosing one vertex, and so on, stopping after as many hops as the student wishes. The student then takes a card from the destination vertex's reserves, thus creating the edge.

In addition to the students representing the graph and carrying out the mutation, one more volunteer is chosen to be the marker. This student attempts to mark all food vertices (by placing their hat on their head) while not marking any vertices that were garbage at the beginning of execution. The naive mark-and-sweep algorithm should be simulated where the marker begins by marking the root, and then recursively marks each vertex in the root's edge cards.

When the marker decides they have completed their task, their work can be checked. One way to do this is to have all students sit down, then perform a depth-first search of the graph beginning with the root and asking each student, as their name is read, to stand up (and read the name on one of their edge cards). At the end, all food vertices are standing and it can be seen whether they have all been marked.

Learning Objective. Operational reasoning is dangerous.

This activity is most effective when used to illustrate an *incorrect* algorithm. The naive algorithm consists of marking the root, then repeatedly examining some marked vertex and marking its neighbors. This algorithm, however, is not guaranteed to mark all food. The run that exposes this error in the algorithm is somewhat pathological and unlikely to arise by random chance. To illustrate the error, then, some collusion among the student volunteers is necessary.

The simplest way to frustrate the marking of all food is to arrange ahead of time with two students to be special vertices, each of which has an edge to the instructor. It is important that (i) no other vertices have edges to the instructor and (ii) both special vertices remain reachable from

the root continually through the activity. The first property is ensured by only giving out two edge cards from the instructor, one to each special vertex. The second property is ensured by making both special vertices immediate neighbors of the root and instructing the root to never discard either of their edge cards.

During the activity, each special vertex keeps the instructor edge card in their hands, unless they are about to be checked by the marking process. Before being checked, they casually discard the instructor edge. After being checked, they can safely pick that card back up. Thus, the only time they are not holding the instructor edge card is when they are being checked by the marker process.

At the end of the activity, the entire class should be surprised that the naive algorithm failed to mark all food. The operational argument, that marks spread throughout the root's connected component, is compelling. This activity is useful, therefore, in illustrating the dangers in anthropomorphic, informal, handwaving arguments.

Tips for success. The activity depends on students knowing each other's names, since names on index cards encode edges. Students should be encouraged to mill around and actively add and discard edges.

The class should not be aware of the collusion between the two students described above, so it should be arranged beforehand. During the activity, asking the root not to discard any edge cards generally does not raise suspicions. To further obfuscate the collusion, the two volunteers should not be physically near each other. The goal is for none of the participants to be aware of the special interleaving of concurrent actions being orchestrated to frustrate the marking of a particular food vertex (the instructor).

In addition to identifying all vertices that are food at the end of the activity, a correct marking algorithm is also required to *not* mark vertices that are initially garbage. A random 3-regular directed graph with 15 vertices, as described in this activity, has an astronomically small probability of containing garbage initially. Therefore, the graph must either be deliberately constructed to contain garbage initially, or a smaller degree must be used[1]. One way to deliberately construct the graph with garbage is to partition the students representing vertices into 2 groups and then distribute edge cards (i) within each group, and (ii) from the group containing the root to the other group (thus forming edges pointing *towards* the group with the root). At the end of the activity, students representing vertices that were initially garbage can be asked to stand to verify that none have been marked.

3.3 Stabilizing Leader Election

One challenge of kinesthetic learning activities is the possibility that things can go amiss during the activity itself. For example, algorithmic simulations can witness communication faults whenever students misunderstand each other's speech or handwriting. Sometimes a single mistake can precipitate a global algorithmic failure. In the examples we have tested, this can usually be avoided by careful design and management of the activity itself. This next activity, however, actually leverages the possibility of faulty executions into a learning opportunity for illustrating the self-healing properties of stabilizing systems. Specifically, this activity shows how the impact of data corruptions can be temporally isolated and repaired during the execution of a stabilizing leader-election algorithm.

The Problem. Each process in a connected graph has a distinct numeric identifier. The goal is to elect as leader the unique process in the system with the greatest identifier. The fault model assumes that process identifiers are not corruptible, but that transient faults may corrupt the data values of all other variables finitely many times during any run. Such faults can be repaired only by overwriting the corrupted values with fresh values.

Description of the activity. This activity simulates two leader election algorithms drawn from [8, pp. 34–36]. The first algorithm is intolerant: It may yield invalid election outcomes in the wake of certain data corruptions. The second is stabilizing: It always detects and recovers from finitely many transient corruptions to non-identifier variables. In both activities, each student represents a process in the graph. Each pair of students with adjacent seats in the classroom shares an undirected edge in the graph.

In principle, each student needs a distinct process identifier; in practice, only the maximal identifier needs to be unique. The identifiers can be assigned by the instructor for finer control of the activity, but this is usually unnecessary. A simpler approach is for each student to use the last four digits of their telephone number.[2]

The Intolerant Algorithm. Each student maintains on a flash card a local variable called *candidate*, which records the maximum identifier witnessed thus far. Upon start-up, each student initializes their candidate variable to the value of their own process identifier. Thereafter, the algorithm proceeds via concurrent gossiping as follows: (1) Every student periodically shares the current value of their candidate variable with each neighbor; (2) After each exchange of information, the value of each local candidate variable gets updated to equal the max of the two values just witnessed.

Learning Objective 1. Single, local faults can precipitate global algorithmic failures.

Not all faults lead to an erroneous result: the spurious candidate value must be globally maximum. One colluding participant can surreptitiously corrupt their candidate value (but not their actual process identifier) to 9999 — a value that is globally maximal and is very unlikely to denote any legitimate process identifier in the system. The corrupted value will diffuse throughout the graph until it ultimately gets elected leader. At this point, the instructor can ask for the person with the elected identifier 9999 to stand up. The absence of such a person witnesses the failure of the algorithm, and provides a basis for classroom discussion about how the intolerant algorithm can be amended.

The Stabilizing Algorithm. This solution recovers from transient data corruptions by recomputing floating outputs — a technique that filters out spurious candidates to prevent phantom leaders from being elected. The key idea is that each legitimate candidate must correspond to some actual, reachable process in the system. For any system with at most $k+1$ processes, the *minimal* path from any node to a legitimate candidate can never exceed k hops. Each node continually recomputes the shortest distance to the maximal candidate value seen thus far, except that candidates more than k hops away are excluded as spurious.

[1] A 2-regular graph with 15 vertices still only has about a 10% chance of containing garbage initially, while a 1-regular graph is virtually guaranteed to contain garbage initially.

[2] Duplicate maximal values are improbable using this assignment strategy, but beware of roommates!

Each student maintains two corruptible variables: *candidate* and *distance*. Each student continuously monitors their neighbor's candidate and distance variables and updates their own values to maintain the following property: Their own candidate value is equal to the maximum of all neighbors' candidates and their own identifier. In addition, their own distance value is one greater than the *minimum* distance value of a neighbor with the *maximum* candidate value (or 0 if the maximum candidate value is their own identifier). It is important that only neighbors with a distance value less than k are considered during this operation.

While in the intolerant algorithm, students use their own candidate values for comparison, it is important that in the stabilizing algorithm one's own candidate and distance values are *not* used in updating this information.

Learning Objective 2. Transient faults can be repaired via computational redundancy.

In the algorithm, any maximal identifier that is spurious will also have a spurious distance. As such, each round of execution will cause the estimated distance to this node to increase. Eventually the estimated distance will exceed a known bound k on the number of nodes in the graph, at which point the spurious identifier is also eliminated from consideration as a candidate for leader. This activity demonstrates how self-repairing algorithms can withstand data corruption using only local communication and coordination.

4. DESIGNING KINESTHETIC LEARNING ACTIVITIES

Developing a KLA requires careful planning. A poorly planned activity will waste class time, and may even be pedagogically harmful, undermining the intended lesson. The design of a KLA should begin with an explicit statement of the learning objectives it is meant to support. The activity should then be designed around these learning objectives.

In addition to this basic principle, the following heuristics are helpful in creating effective KLAs.

- Incorporate an element of surprise. With parallel garbage collection, students are surprised when the "obvious" algorithm does not produce the correct result.
- Involve multiple senses and dimensions of engagement. In the leader election activity, students are observing neighbor values while modifying their own.
- Anticipate and accommodate mistakes. There will almost certainly be too many concurrent activities to be able to monitor them all. The activity should either be robust enough to tolerate the occasional mistake, or checks should be incorporated to reduce the chances of such a mistake occurring.
- Engage the entire class. If the activity can not be scaled to include every student directly, it should be designed to encourage non-participants to identify with participants, and thus be involved at least vicariously.
- Provide simple directions to participants. If the instructions are complicated, there is a greater chance of mistakes being made. Also, if students are too engrossed in their local computation, the bigger picture can be hard for them to see.

Even if an activity has been carefully planned, a practice run is an invaluable aid in assessing how it will work in a classroom setting. A small, friendly group can provide feedback and insight into an activity's dynamics and can help refine the details of the presentation to improve its chances of success in the classroom.

5. CONCLUSIONS

KLAs promote student interactivity and improve student learning by engaging a fundamental and ubiquitous learning style. For this reason, courses across all disciplines can benefit from the inclusion of such activities. Courses on distributed computing, however, are ideally suited for KLAs since they naturally incorporate concurrency and locality of computation. Self-stabilization improves the robustness of KLAs to participant error.

We have provided here a collection of KLAs developed for use in our courses on distributed computing. Our hope is that others will adopt these activities for their own courses, as well as use this collection as a template for developing new activities that they later share with the community, too.

6. REFERENCES

[1] D. P. Ausubel, J. D. Novak, and H. Hanesian. *Educational Psychology: A Cognitive View*. Holt, Rinehart and Winston, 2nd edition, 1978.

[2] A. Begel, D. D. Garcia, and S. A. Wolfman. Kinesthetic learning in the classroom. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 183–184, New York, NY, USA, 2004. ACM Press.

[3] B. S. Bloom, M. D. Engelhart, H. H. Hill, E. J. Furst, and D. R. Krathwhol, editors. *Taxonomy of Educational Objectives. The Classification of Educational Goals, Handbook I: Cognitive Domain*. David McKay Company, Inc, New York, 1956.

[4] C. C. Bonwell. Enhancing the lecture: Revitalizing the traditional format. *New Directions for Teaching and Learning*, 67:31–44, Fall 1996.

[5] P. Bucci, T. J. Long, B. W. Weide, and J. Hollingsworth. Toys are us: Presenting mathematical concepts in CS1/CS2. In *Proceedings of the* 30^{th} *ASEE/IEEE Frontiers in Education Conference*. IEEE Computer Society Press, 2000.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[7] E. W. Dijkstra. On anthropomorphism in science. EWD936, Sept. 1985.

[8] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[9] C. G. Jung. *Psychological Types*. Routledge, 1992. Original work published in 1921.

[10] H. L. Lujan and S. E. DiCarlo. First-year medical students prefer multiple learning styles. *Advances in Physiology Education*, 30(1):13–16, March 2006.

[11] I. B. Myers, M. H. McCaulley, N. L. Quenk, and A. L. Hammer. *MBTI Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator*. Consulting Psychologists Press, 3rd edition, 1998.