ZDD

counting graph partitions. quickly?

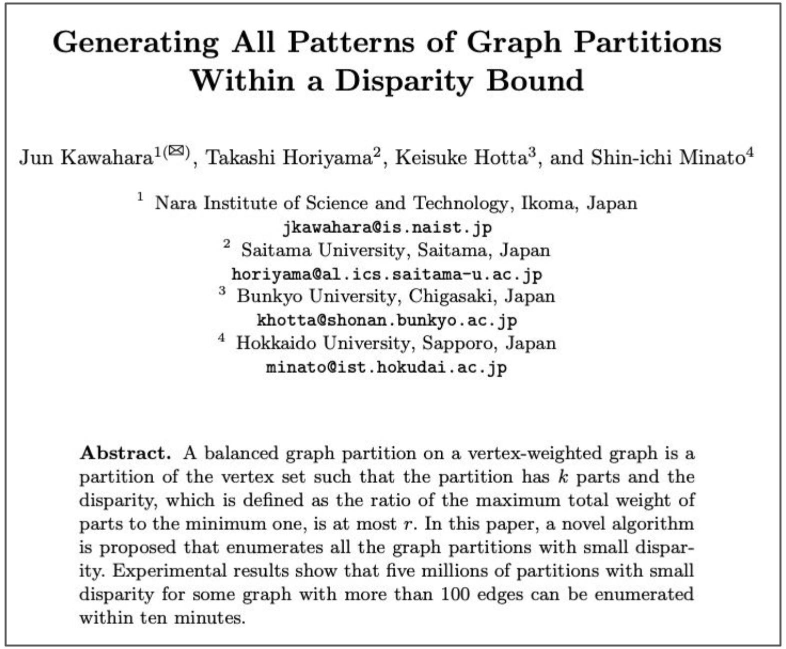
Gabe Schoenbach, Bhushan Suwal, Amy Becker
MGGG | Feb. 2021

interesting questions

- how many ways can we partition the 10x10 grid into 10 equal-sized parts?
  - $|V| = 100$
  - $|E| = 180$
  - $k = 10$, each part contains precisely 10 nodes
- how many ways can we split Iowa into 4 equal-sized CDs?
  - $|V| = 99$
  - $|E| = 222$
  - $k = 4$, each part deviates by at most 1% from the ideal population

introducing the enumpart algorithm

- uses the ZDD data structure to count ALL the ways to partition a graph into *k* parts
  - *does not* care about size/weight of parts
- we can sample uniformly and get an exact count quickly
- ...but slower to save each partition to disk
- implemented in C++ (available in Python via **graphillion** “soon”)
- we replicated our own version of enumpart in Python and Julia



Generating All Patterns of Graph Partitions Within a Disparity Bound

Jun Kawahara[1](✉), Takashi Horiyama[2], Keisuke Hotta[3], and Shin-ichi Minato[4]

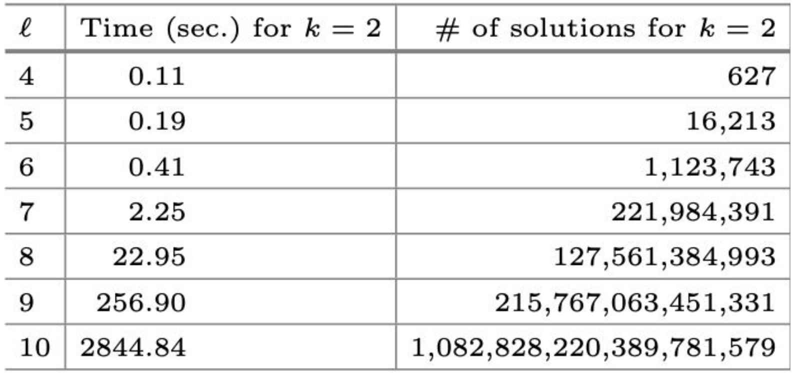
[1] Nara Institute of Science and Technology, Ikoma, Japan
jkawahara@is.naist.jp
[2] Saitama University, Saitama, Japan
horiyama@al.ics.saitama-u.ac.jp
[3] Bunkyo University, Chigasaki, Japan
khotta@shonan.bunkyo.ac.jp
[4] Hokkaido University, Sapporo, Japan
minato@ist.hokudai.ac.jp

Abstract. A balanced graph partition on a vertex-weighted graph is a partition of the vertex set such that the partition has k parts and the disparity, which is defined as the ratio of the maximum total weight of parts to the minimum one, is at most r. In this paper, a novel algorithm is proposed that enumerates all the graph partitions with small disparity. Experimental results show that five millions of partitions with small disparity for some graph with more than 100 edges can be enumerated within ten minutes.

Kawahara et al [2017]

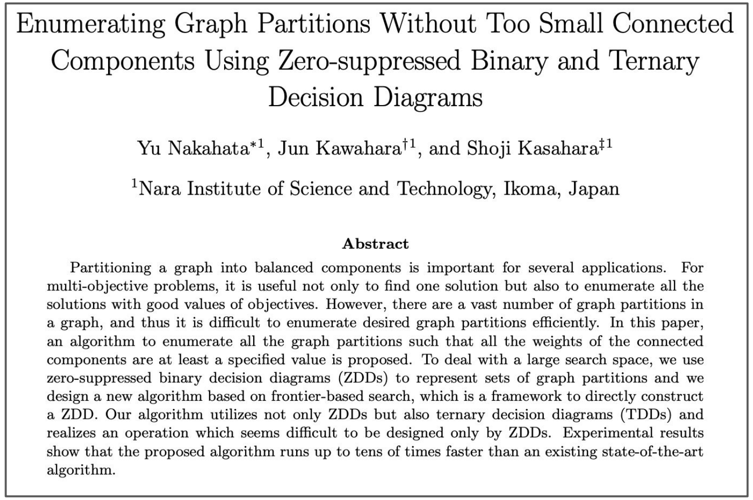
enumpart with weights

- disparity = $max_{i,j\in[k]}\left\{\frac{w(p_i)}{w(p_j)}\right\}$ where p_r is a part in our partition, $w(p_r)$ is the weight
  - perfectly balanced partition → disparity = 1
- adding weights requires more data at each step, dramatically slowing down the computation
- future work: julia-ify their pseudocode →



ℓ	Time (sec.) for $k = 2$	# of solutions for $k = 2$
4	0.11	627
5	0.19	16,213
6	0.41	1,123,743
7	2.25	221,984,391
8	22.95	127,561,384,993
9	256.90	215,767,063,451,331
10	2844.84	1,082,828,220,389,781,579

performance on ℓxℓ grid graph into 2 (no weights)



Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams

Yu Nakahata*1, Jun Kawahara†1, and Shoji Kasahara‡1

1Nara Institute of Science and Technology, Ikoma, Japan

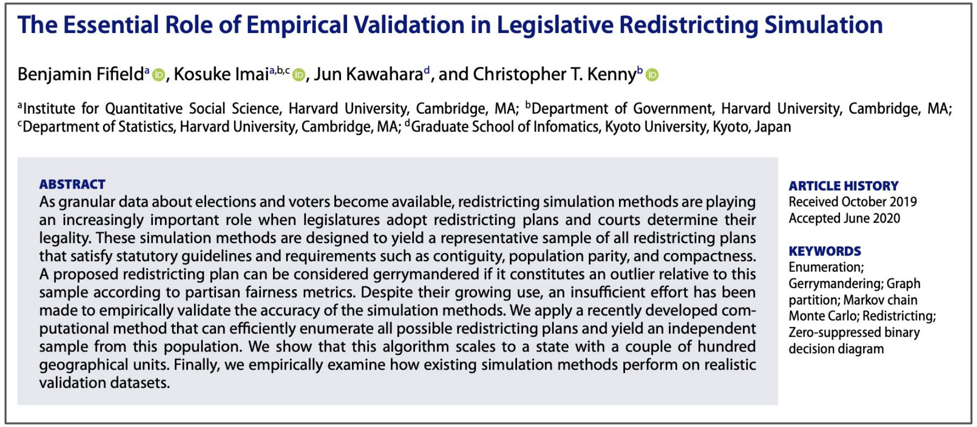
Abstract

Partitioning a graph into balanced components is important for several applications. For multi-objective problems, it is useful not only to find one solution but also to enumerate all the solutions with good values of objectives. However, there are a vast number of graph partitions in a graph, and thus it is difficult to enumerate desired graph partitions efficiently. In this paper, an algorithm to enumerate all the graph partitions such that all the weights of the connected components are at least a specified value is proposed. To deal with a large search space, we use zero-suppressed binary decision diagrams (ZDDs) to represent sets of graph partitions and we design a new algorithm based on frontier-based search, which is a framework to directly construct a ZDD. Our algorithm utilizes not only ZDDs but also ternary decision diagrams (TDDs) and realizes an operation which seems difficult to be designed only by ZDDs. Experimental results show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.

Nakahata et. al [2018]

imai’s paper

- uses **enumpart** to generate a set of contiguous districting plans
  - either *all* contiguous plans, or a uniformly sampled subset of all contiguous plans
- then, winnow those sets down to a smaller ensemble of plans that satisfy some population and compactness constraints
- compare those ensembles to those found by MCMC



The Essential Role of Empirical Validation in Legislative Redistricting Simulation

Benjamin Fifield[a], Kosuke Imai[a,b,c], Jun Kawahara[d], and Christopher T. Kenny[b]

[a]Institute for Quantitative Social Science, Harvard University, Cambridge, MA; [b]Department of Government, Harvard University, Cambridge, MA; [c]Department of Statistics, Harvard University, Cambridge, MA; [d]Graduate School of Infomatics, Kyoto University, Kyoto, Japan

ABSTRACT
As granular data about elections and voters become available, redistricting simulation methods are playing an increasingly important role when legislatures adopt redistricting plans and courts determine their legality. These simulation methods are designed to yield a representative sample of all redistricting plans that satisfy statutory guidelines and requirements such as contiguity, population parity, and compactness. A proposed redistricting plan can be considered gerrymandered if it constitutes an outlier relative to this sample according to partisan fairness metrics. Despite their growing use, an insufficient effort has been made to empirically validate the accuracy of the simulation methods. We apply a recently developed computational method that can efficiently enumerate all possible redistricting plans and yield an independent sample from this population. We show that this algorithm scales to a state with a couple of hundred geographical units. Finally, we empirically examine how existing simulation methods perform on realistic validation datasets.

ARTICLE HISTORY
Received October 2019
Accepted June 2020

KEYWORDS
Enumeration; Gerrymandering; Graph partition; Markov chain Monte Carlo; Redistricting; Zero-suppressed binary decision diagram

Fifield, Imai, Kawahara, Kenny [2020]

imai’s paper

FL — 70 precincts, *k* = 2 **44,082,156** total *k*-partitions	**FL — 250** precincts, *k* = 2 **~10^{39}** total *k*-partitions	**IA — 99** counties, *k* = 4 **~10^{24}** total *k*-partitions
full enumeration 44,082,156 plans time: ~8 hrs	uniform sampling 100,000,000 plans time: ?	uniform sampling 500,000,000 plans time: ~36 hrs
1% pop. deviation 717,060 plans (1.6%) + compactness constraint 271,240 plans (0.6%)	1% pop. deviation 1,950,000 plans (1.95%)	1% pop. deviation 300(!) plans (0.00006%)

imai’s paper

FL — 70 precincts, k = 2 **44,082,156** total k-partitions	**FL — 250** precincts, k = 2 **~10^{39}** total k-partitions	**IA — 99** counties, k = 4 **~10^{24}** total k-partitions
full enumeration 44,082,156 plans time: ~8 hrs	uniform sampling 100,000,000 plans time: ?	uniform sampling 500,000,000 plans time: ~36 hrs
1% pop. deviation 717,060 plans (1.6%) + compactness constraint 271,240 plans (0.6%)	1% pop. deviation 1,950,000 plans (1.95%)	1% pop. deviation 300(!) plans (0.00006%)

~10^{17} total population balanced plans in IA?

imai’s paper

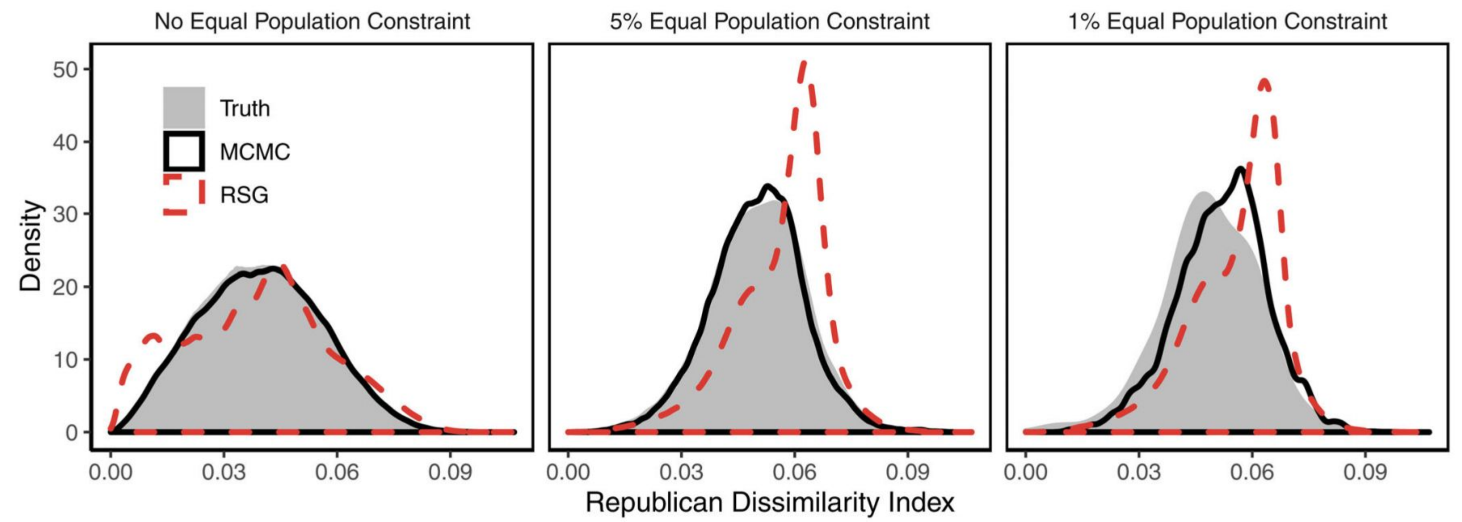
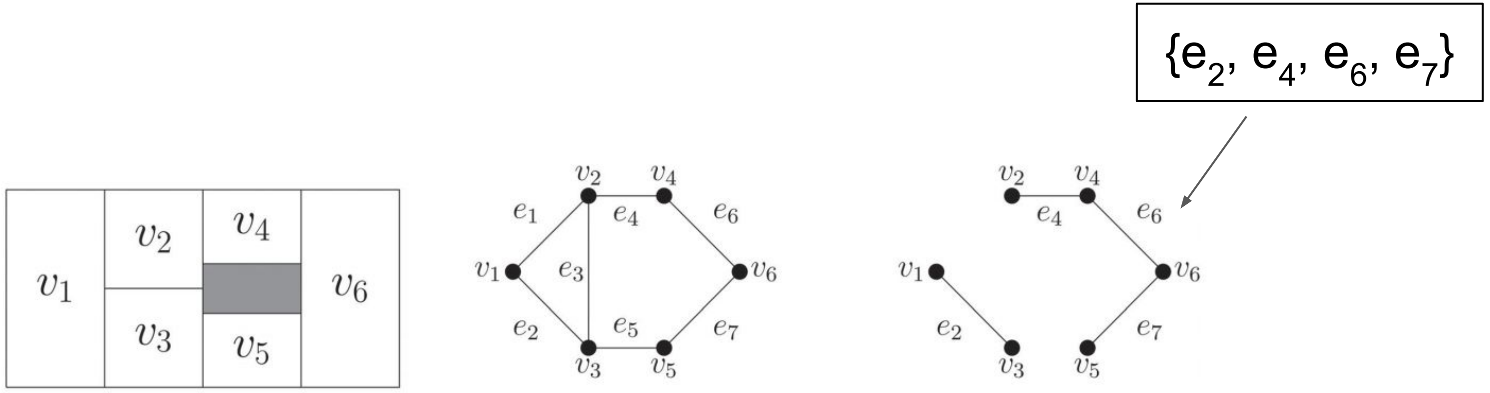


Figure 11. A validation study, uniformly sampling from the population of all partitions of the Iowa map into four districts. The underlying data are Iowa's county map in the left plot of Figure 10, which is partitioned into four congressional districts. As in the previous validation exercises, the Markov chain Monte Carlo (MCMC) method (solid black line) is able to approximate the independently and uniformly sampled target distribution, while the random-seed-and-grow (RSG) method (red dashed line) performs poorly.

how does enumpart work?

- challenge is memory management
- ZDD = Zero-Suppressed Binary Decision Diagram
- goal: efficiently represent families of sets (edges of original graph)

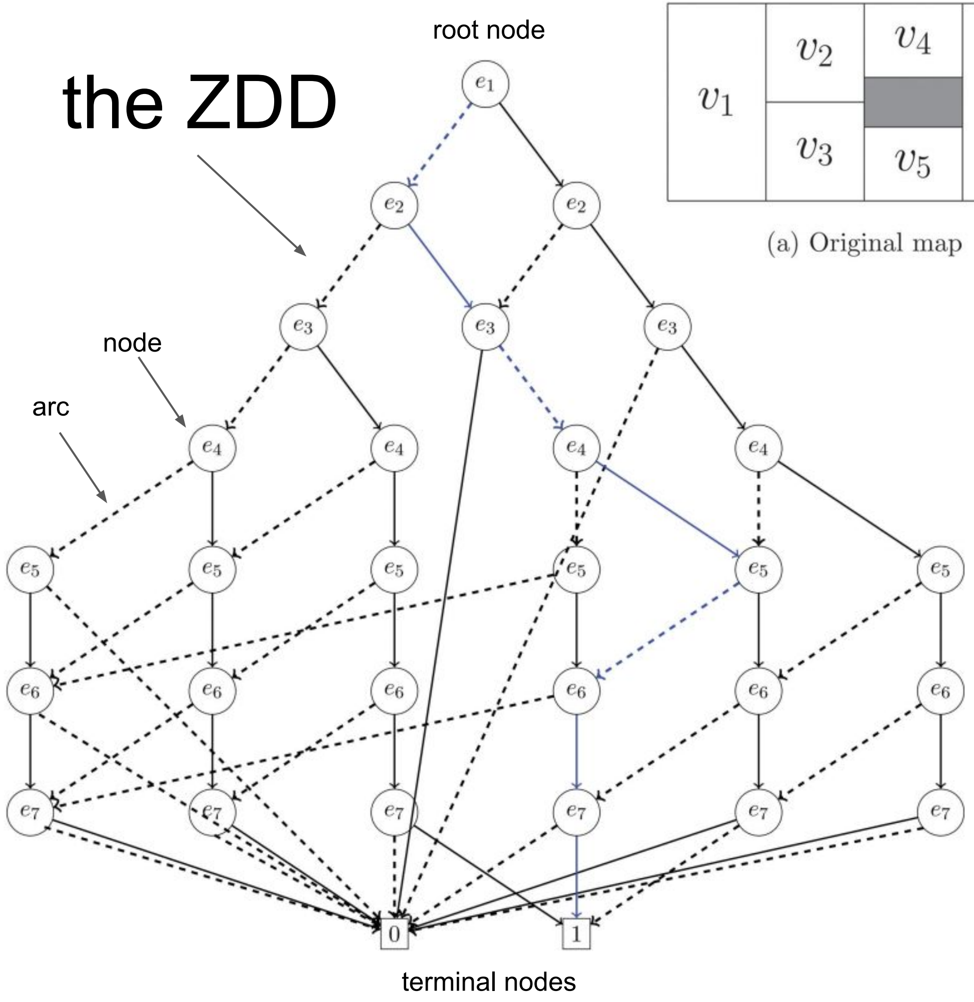
$\{e_2, e_4, e_6, e_7\}$

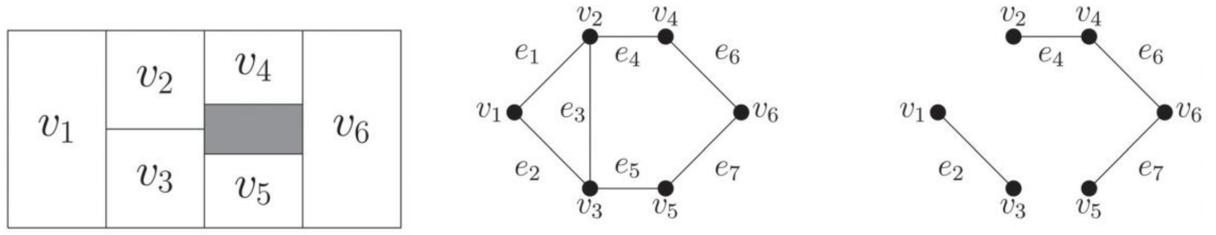


(a) Original map (b) Graph representation (c) Induced subgraph

how does enumpart work?

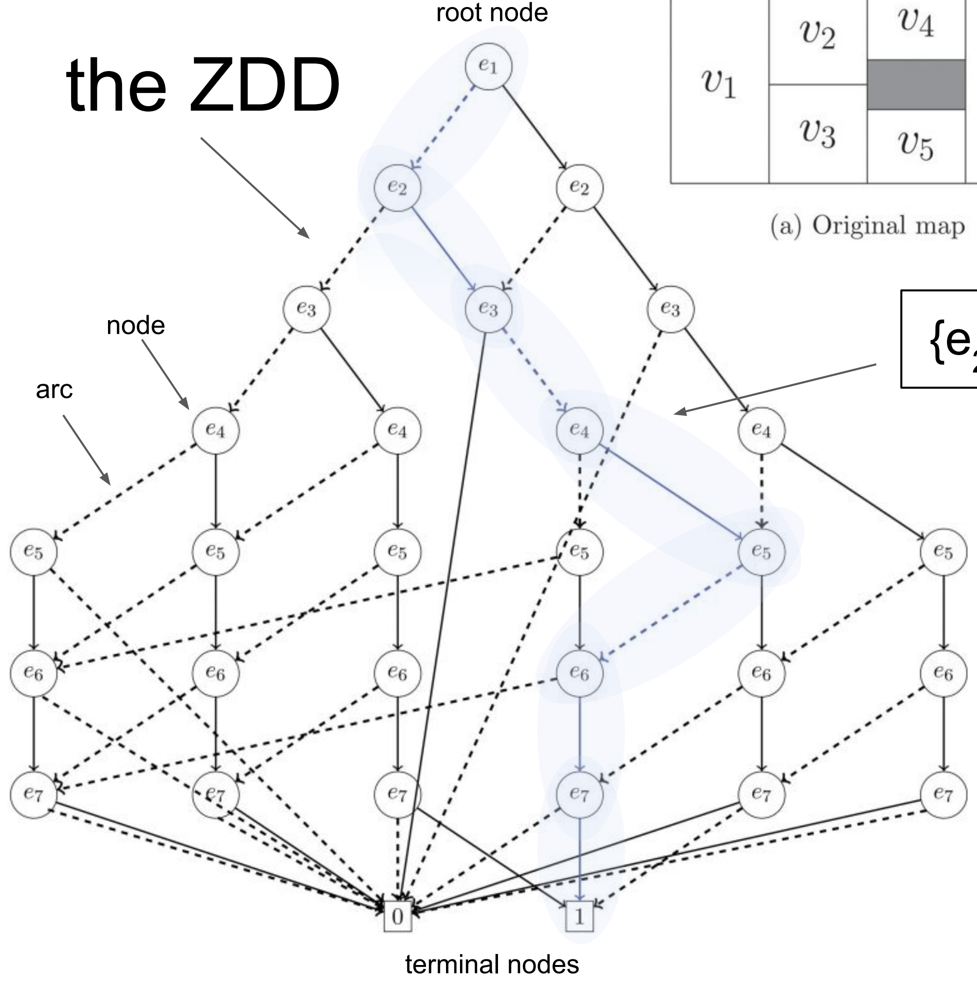
- the ZDD
- how to enumerate
- how to random sample
- constructing the ZDD
  - determining connected components
  - induced subgraph condition
  - *not covered*: accounting for population
- optimizing the ZDD
  - merging nodes
  - ordering edges
- scalability

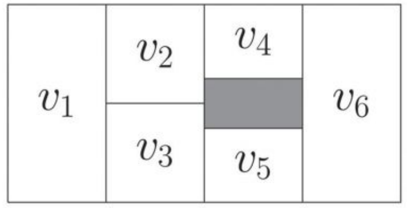




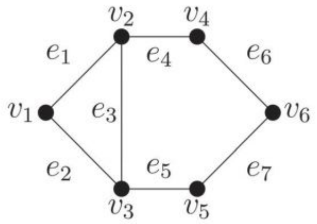
(a) Original map (b) Graph representation (c) Induced subgraph

the ZDD

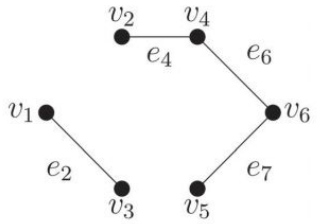




(a) Original map



(b) Graph representation

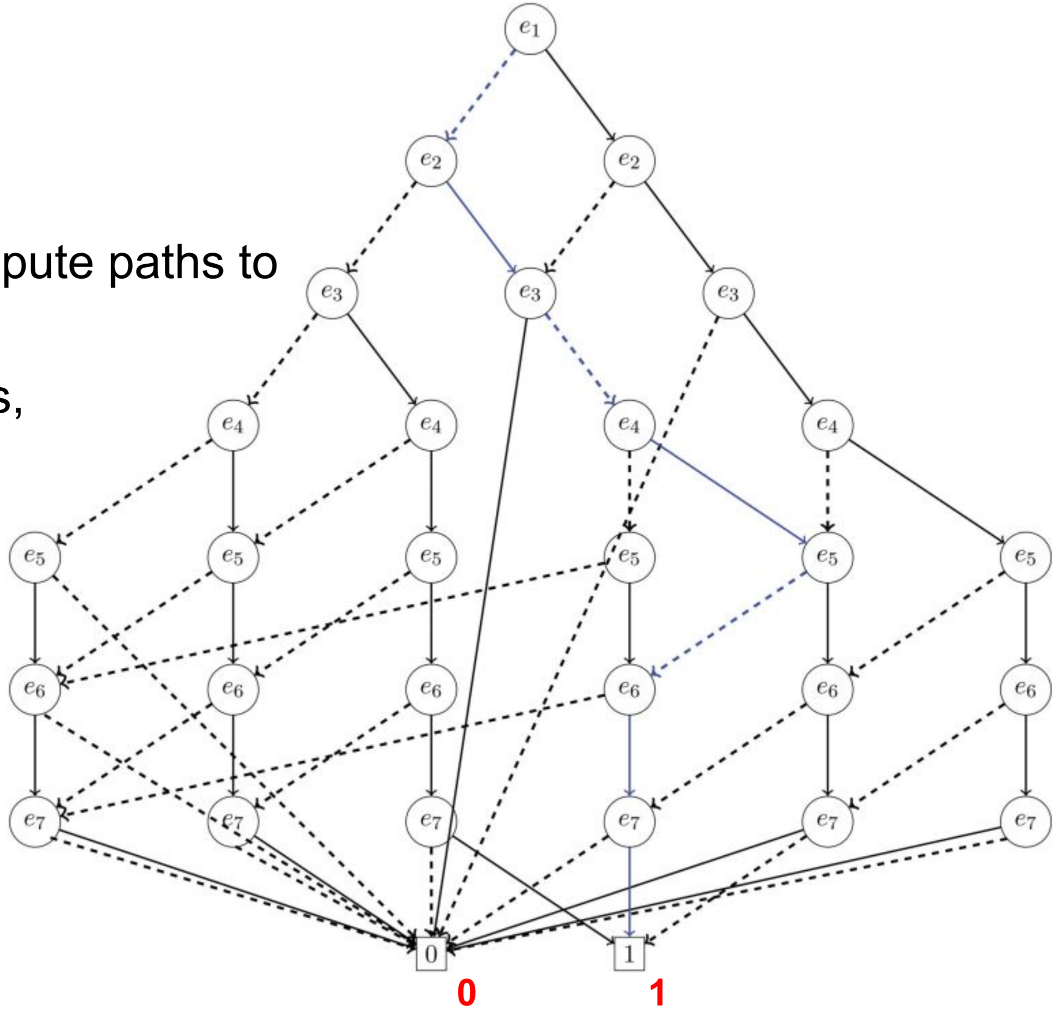


(c) Induced subgraph

$\{e_2, e_4, e_6, e_7\}$

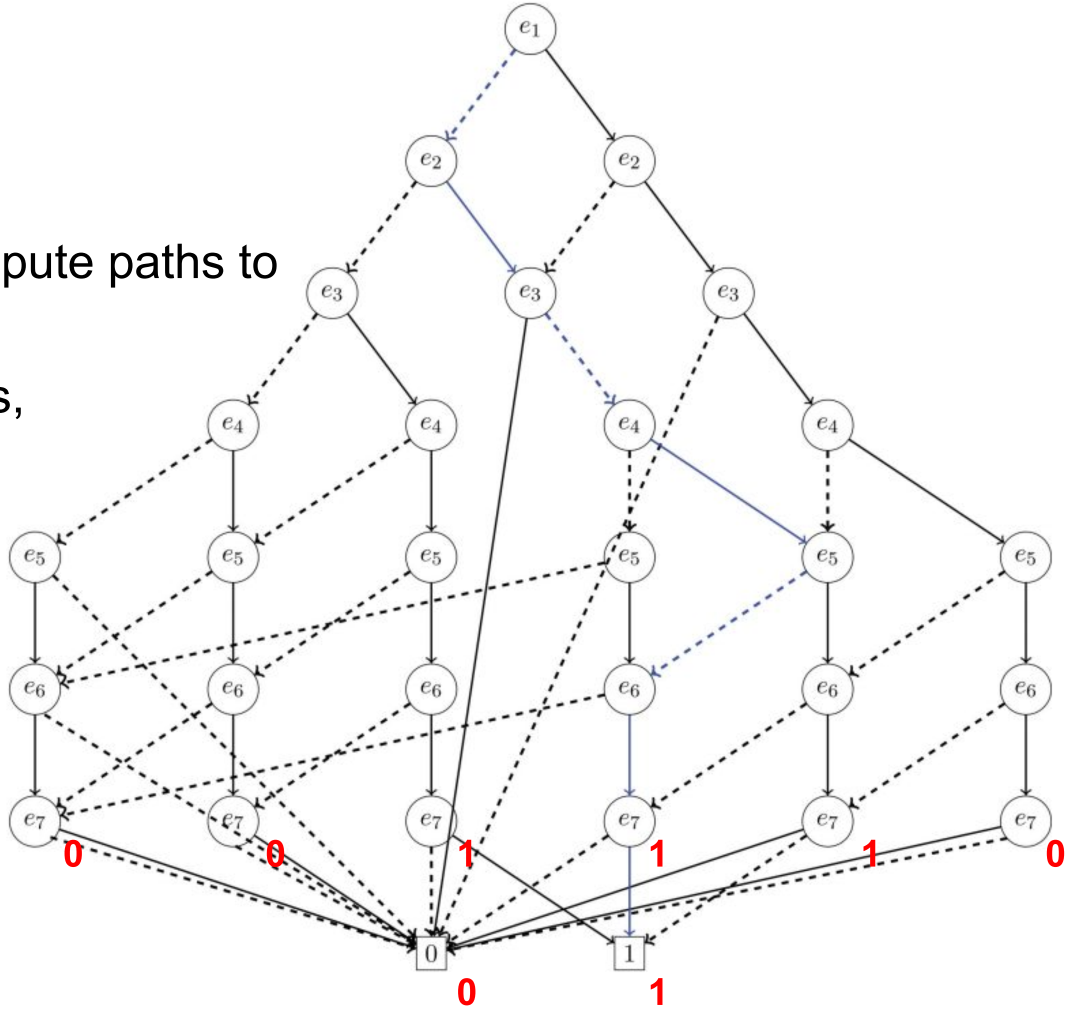
enumeration

- For each node in ZDD, compute paths to the 1-terminal
- Start with 0- and 1-terminals, which have 0 and 1 paths
- Each parent's paths are the sum of children's paths



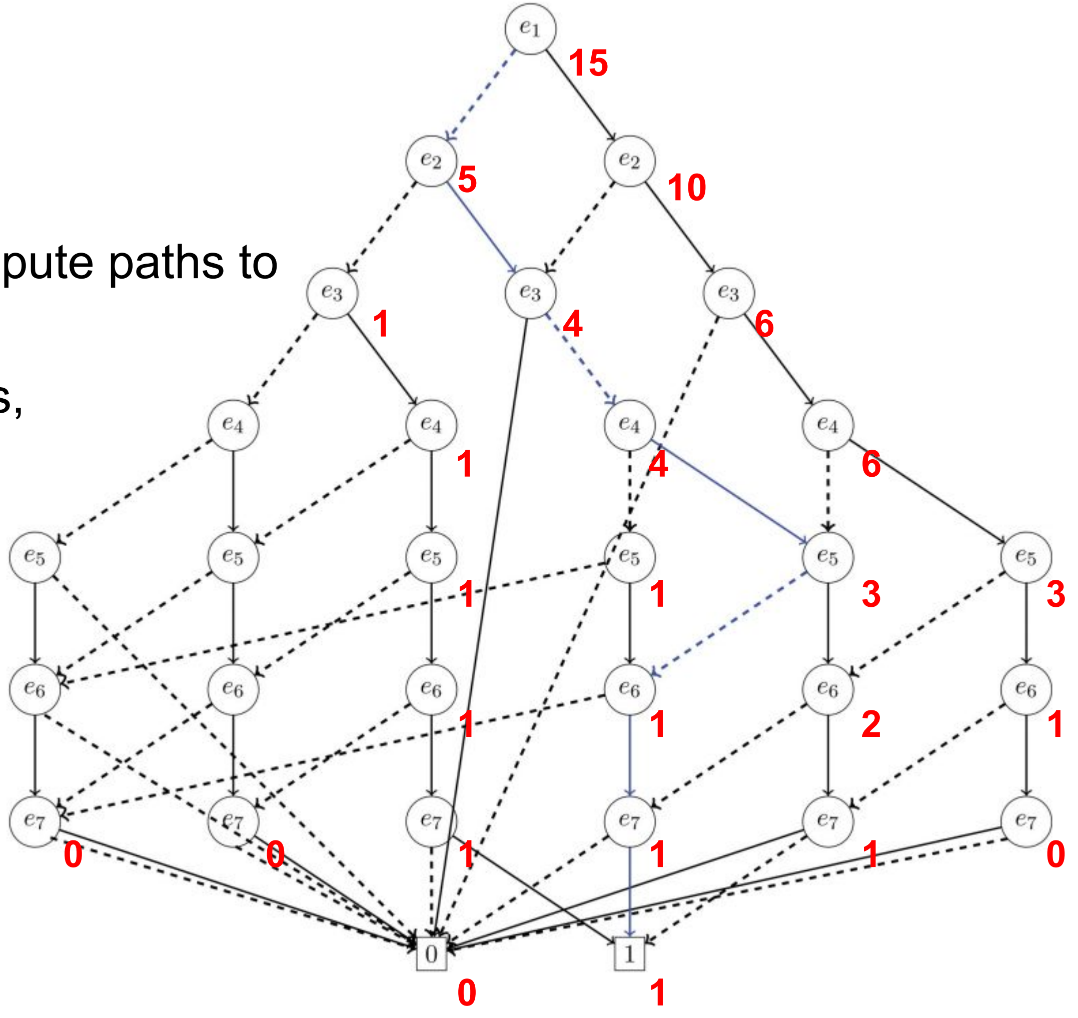
enumeration

- For each node in ZDD, compute paths to the 1-terminal
- Start with 0- and 1-terminals, which have 0 and 1 paths
- Each parent's paths are the sum of children's paths



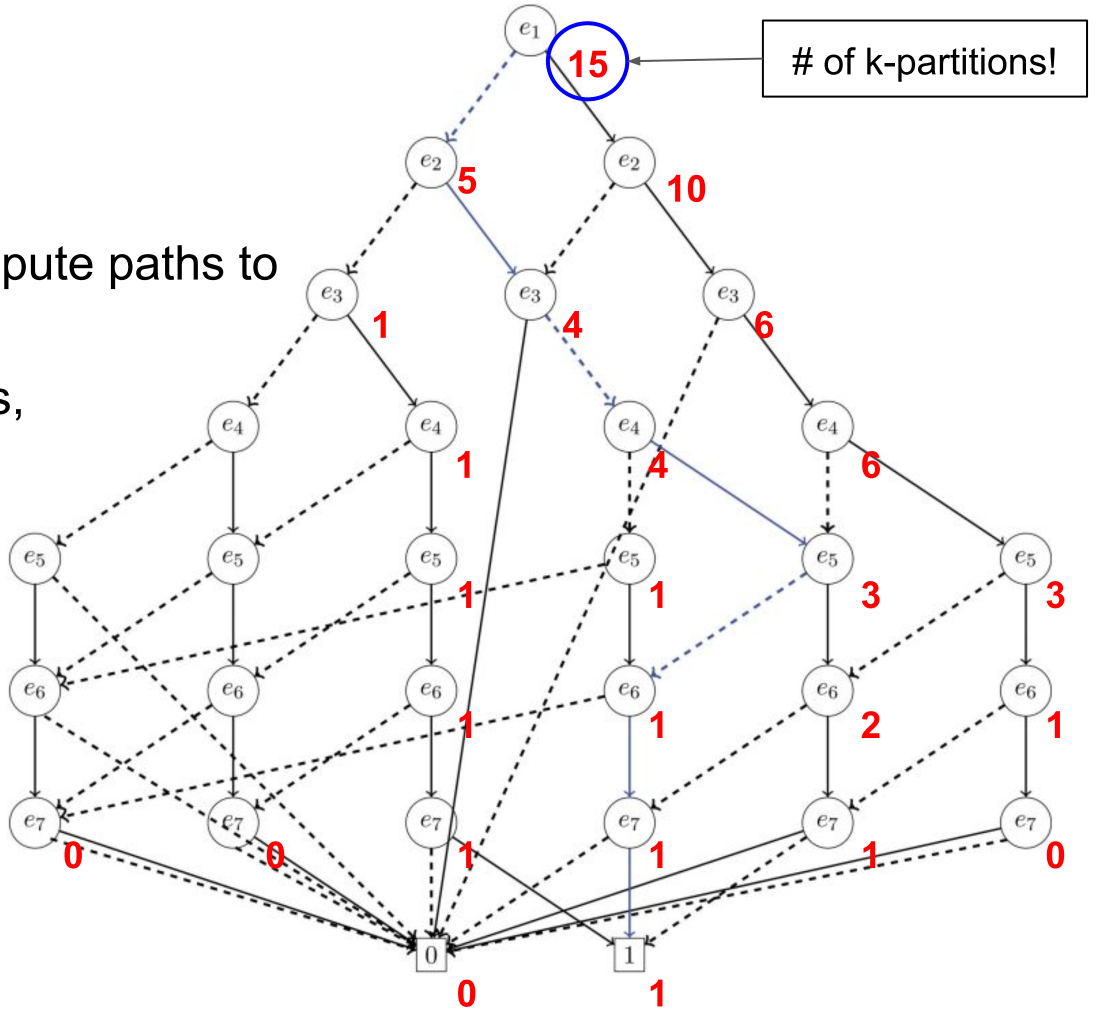
enumeration

- For each node in ZDD, compute paths to the 1-terminal
- Start with 0- and 1-terminals, which have 0 and 1 paths
- Each parent's paths are the sum of children's paths



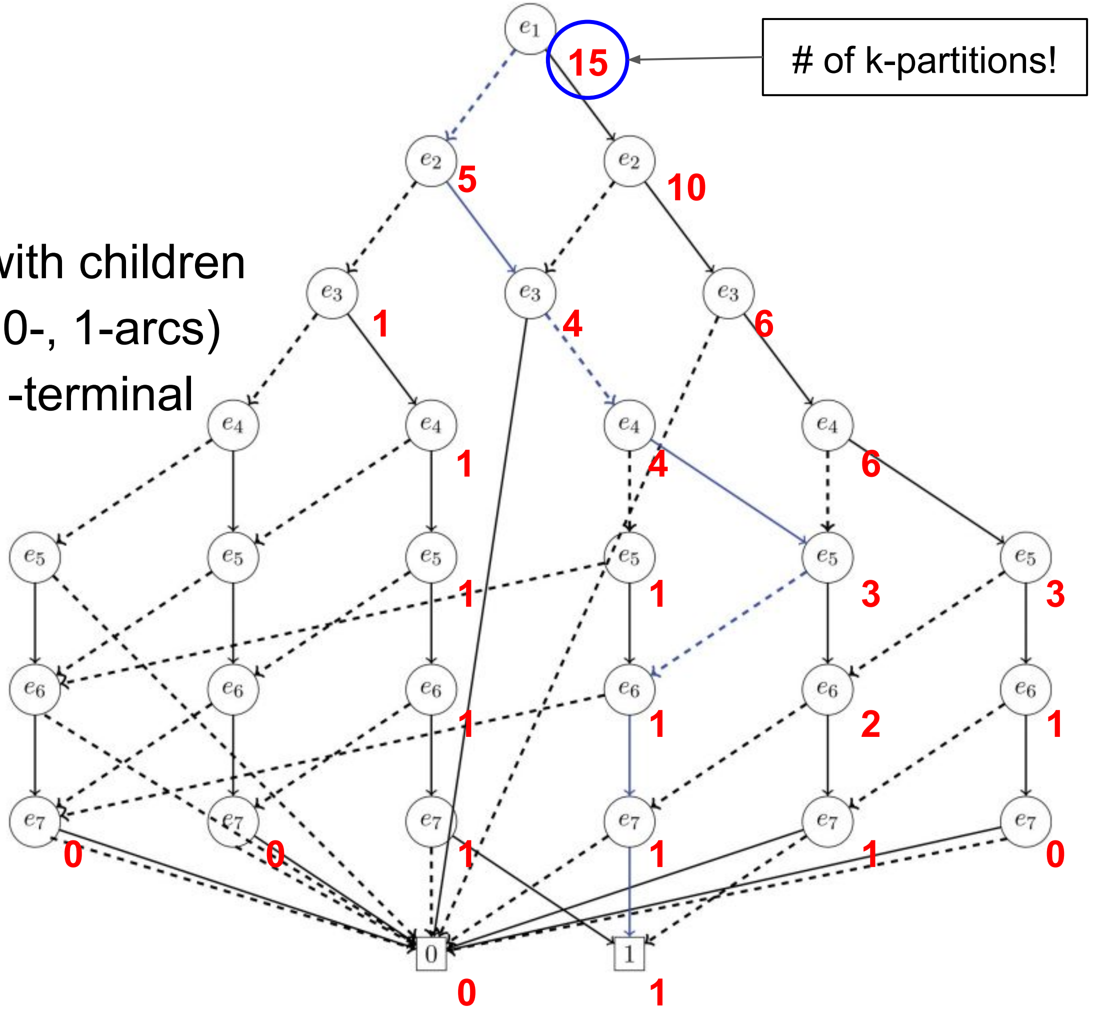
enumeration

- For each node in ZDD, compute paths to the 1-terminal
- Start with 0- and 1-terminals, which have 0 and 1 paths
- Each parent's paths are the sum of children's paths



random sampling

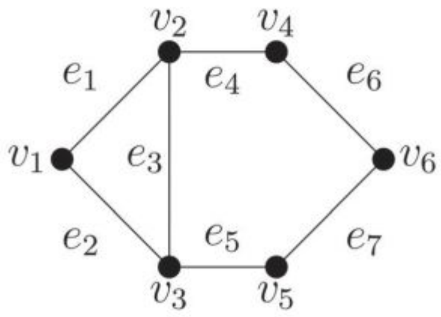
- given a node *v* in the ZDD with children v_0 and v_1 (corresponding to 0-, 1-arcs)
- let *c(v)* = # paths from v to 1-terminal
- $c(v) = c(v_0) + c(v_1)$
- Start with root node, choose node v_1 with probability $\frac{c(v_1)}{c(v_0) + c(v_1)}$



constructing the ZDD

given $G = (V, E)$, we cycle through each edge and decide whether or not to retain it. as we process each edge, we ask ourselves:

- how many connected components have we created?
- have we violated the induced subgraph condition?
- how heavy are the connected components we've created? [if tracking population]

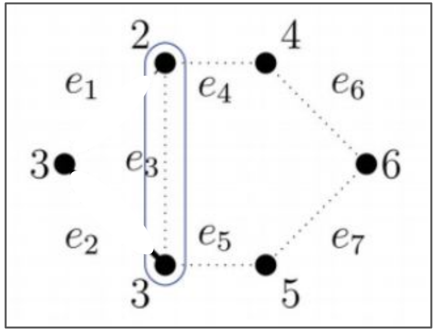


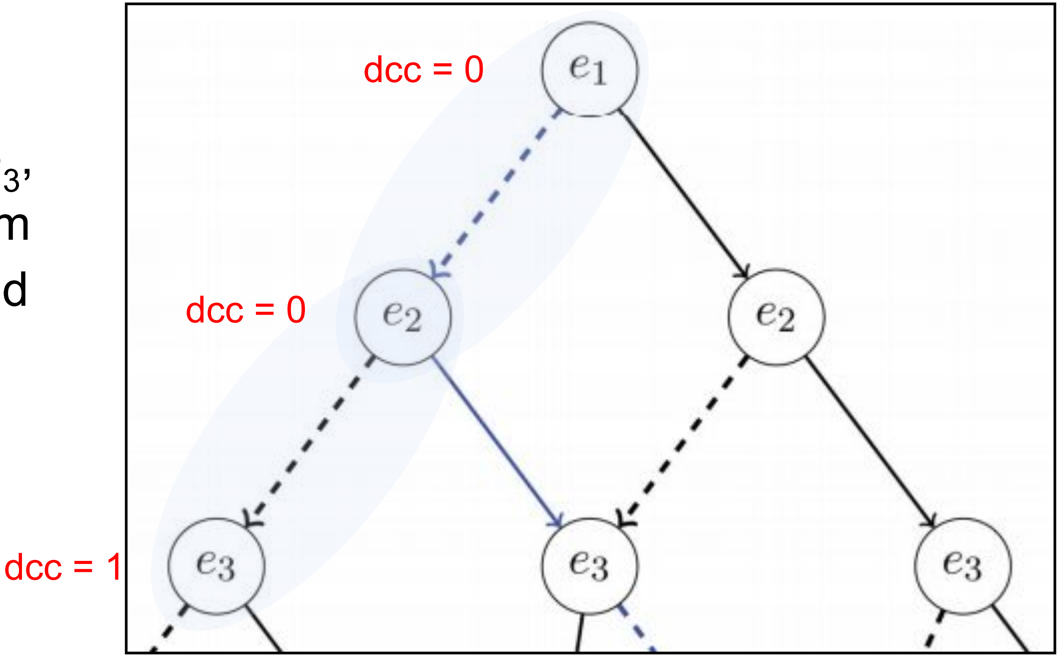
(b) Graph representation

determining the number of connected components

consider e_1 -- > e_2 -- > e_3

regardless of whether we retain e_3, vertex v_1 will be disconnected from the rest of the graph, so we should set **dcc = 1**





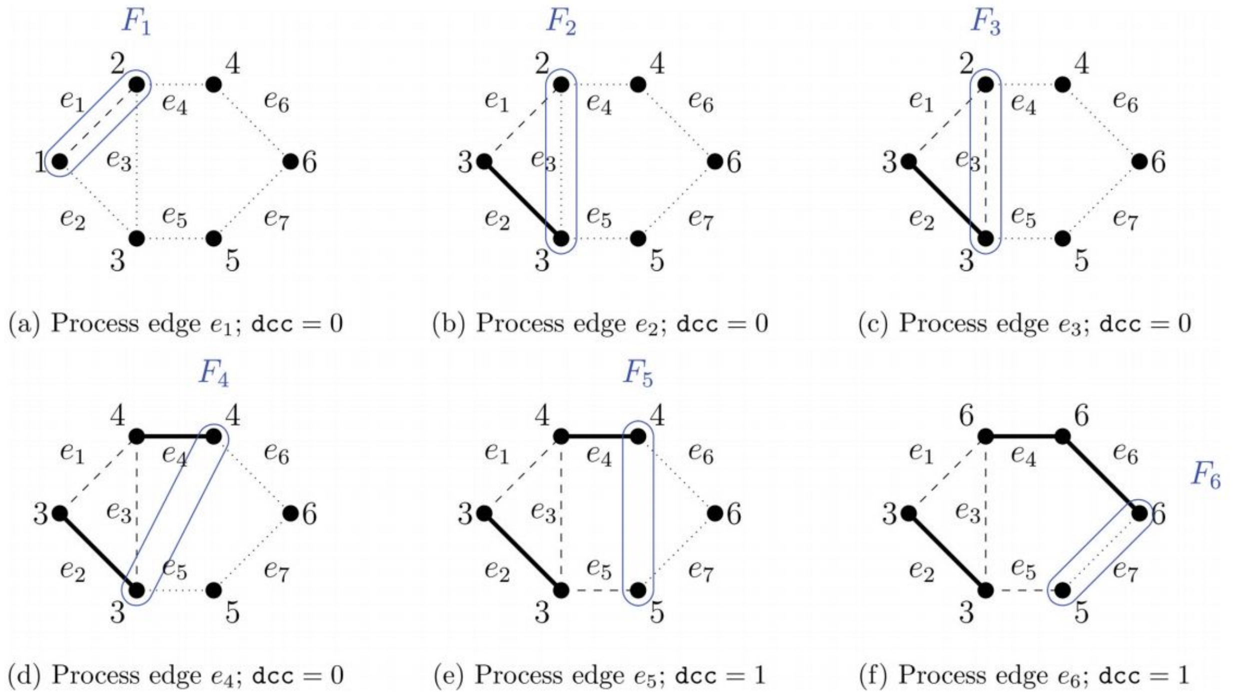
determining the number of connected components

the **connected component number** (**ccn**) is the largest vertex index of all vertices in a connected component

the **frontier** is the set of all vertices which are incident to both a processed and an unprocessed edge.

increment dcc if:

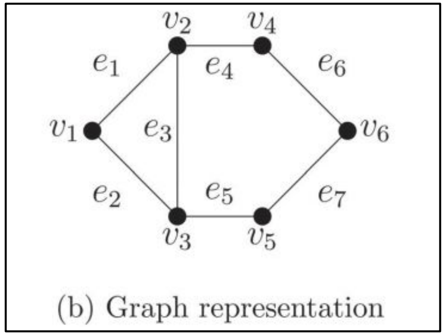
- a vertex *v* has just left the frontier
- *v* is not connected to the frontier



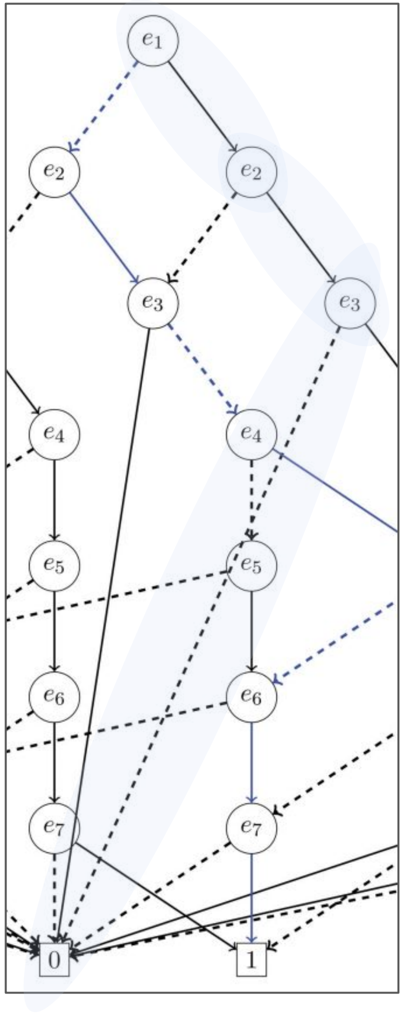
(a) Process edge e_1; $\texttt{dcc} = 0$
(b) Process edge e_2; $\texttt{dcc} = 0$
(c) Process edge e_3; $\texttt{dcc} = 0$
(d) Process edge e_4; $\texttt{dcc} = 0$
(e) Process edge e_5; $\texttt{dcc} = 1$
(f) Process edge e_6; $\texttt{dcc} = 1$

Figure 3. Calculation of the frontier, the connected component number, and the determined connected components. This illustrative example is based on the redistricting problem shown in Figure 1. A positive integer placed next to each vertex represents the connected component number, whereas the vertices grouped by the solid blue line represent a frontier. A connected component is determined when processing edge e_5.

induced subgraph condition



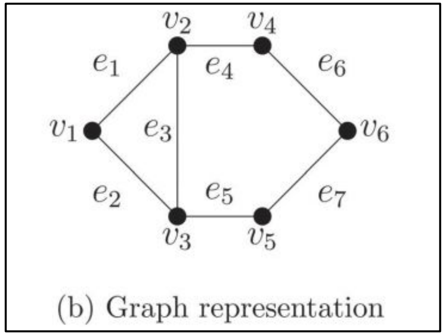
(b) Graph representation



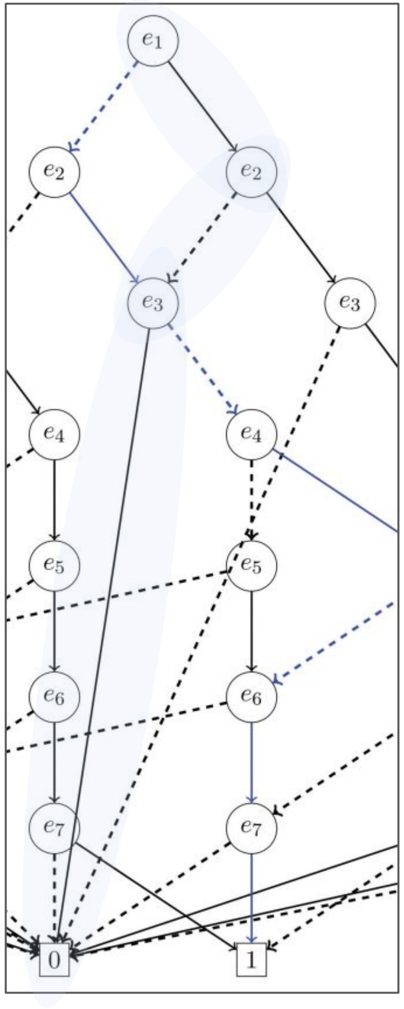
consider $e_1 \longrightarrow e_2 \longrightarrow e_3$

since, in our original graph, e_1, e_2, e_3 are all connected, we *must* also retain e_3, or we will no longer be creating an induced subgraph — therefore, we have $e_1 \longrightarrow e_2 \longrightarrow e_3 \dashrightarrow 0$

induced subgraph condition



(b) Graph representation



consider $e_1 \longrightarrow e_2 \longrightarrow e_3$

since, in our original graph, e_1, e_2, e_3 are all connected, we *must* also retain e_3, or we will no longer be creating an induced subgraph — therefore, we have $e_1 \longrightarrow e_2 \longrightarrow e_3 \dashrightarrow 0$

consider $e_1 \longrightarrow e_2 \dashrightarrow e_3$

now we *cannot* retain e_3, so we have $e_1 \longrightarrow e_2 \dashrightarrow e_3 \longrightarrow 0$

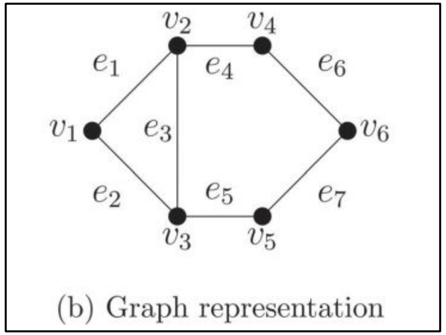
induced subgraph condition

we need to introduce the **forbidden pair set (fps)**

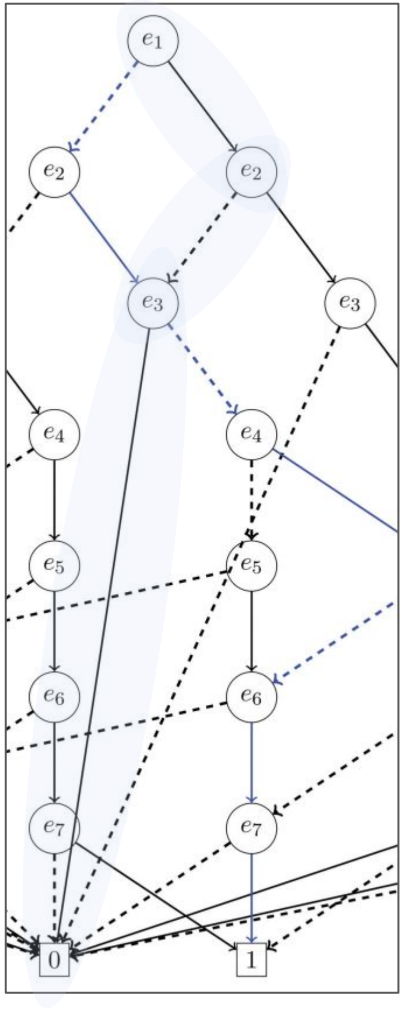
once we decide not to use an edge (like e_2) that connects two distinct components, we can never connect those components

so after e_1 —> e_2 --> e_3, we add {2, 3} to **fps**

when processing e_3, we see that retaining it would connect 2 to 3, so we send e_3 —> 0

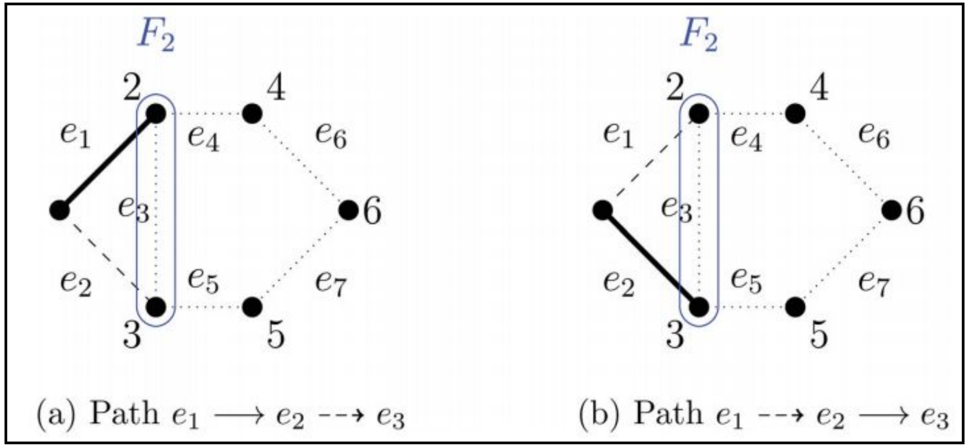


(b) Graph representation



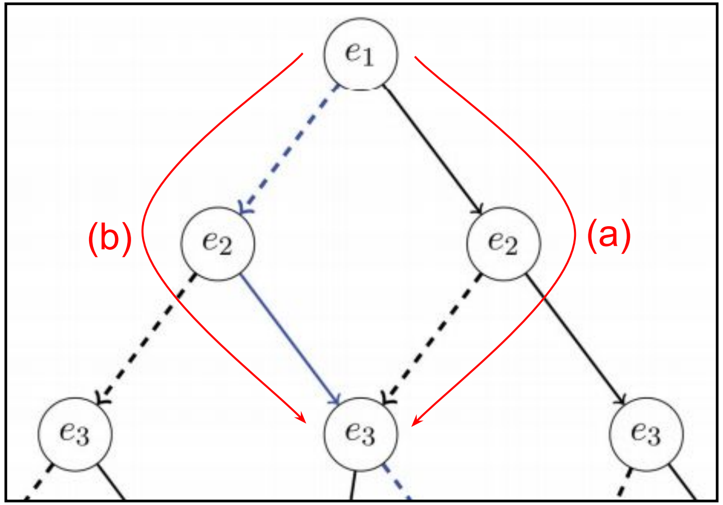
merging nodes

when processing e_j, the only info we need is the connectivity of vertices in F_{j-1}



(a) Path $e_1 \longrightarrow e_2 \dashrightarrow e_3$ (b) Path $e_1 \dashrightarrow e_2 \longrightarrow e_3$

- only need to track the **ccn**'s of vertices in F_{j-1}
- if two nodes that represent e_j have identical **dcc**'s and **ccn**'s on F_{j-1}, we can merge them into one node



ordering edges

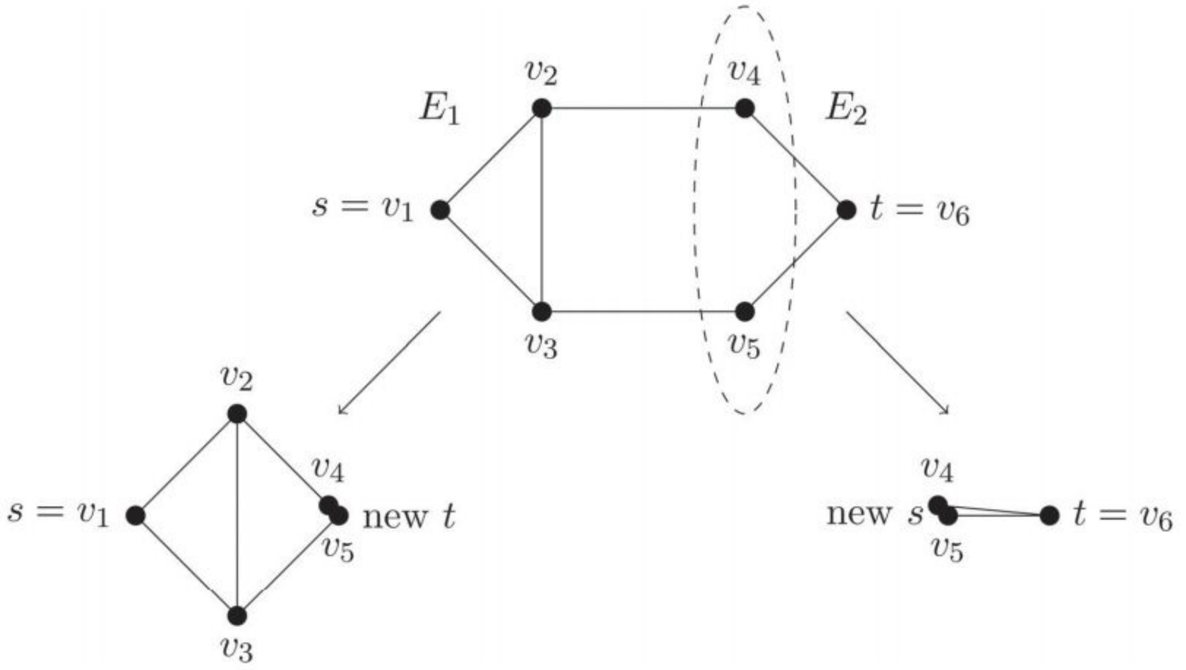


Figure 5. An example of edge ordering by vertex cuts. To order edges, we choose two vertices with the maximum shortest distance and call them *s* and *t*. We then use the minimum vertex cut, indicated by the dashed oval, to create two or more connected components, which are arbitrarily ordered. The same procedure is then applied to each connected component until the resulting connected components are sufficiently small.

scalability

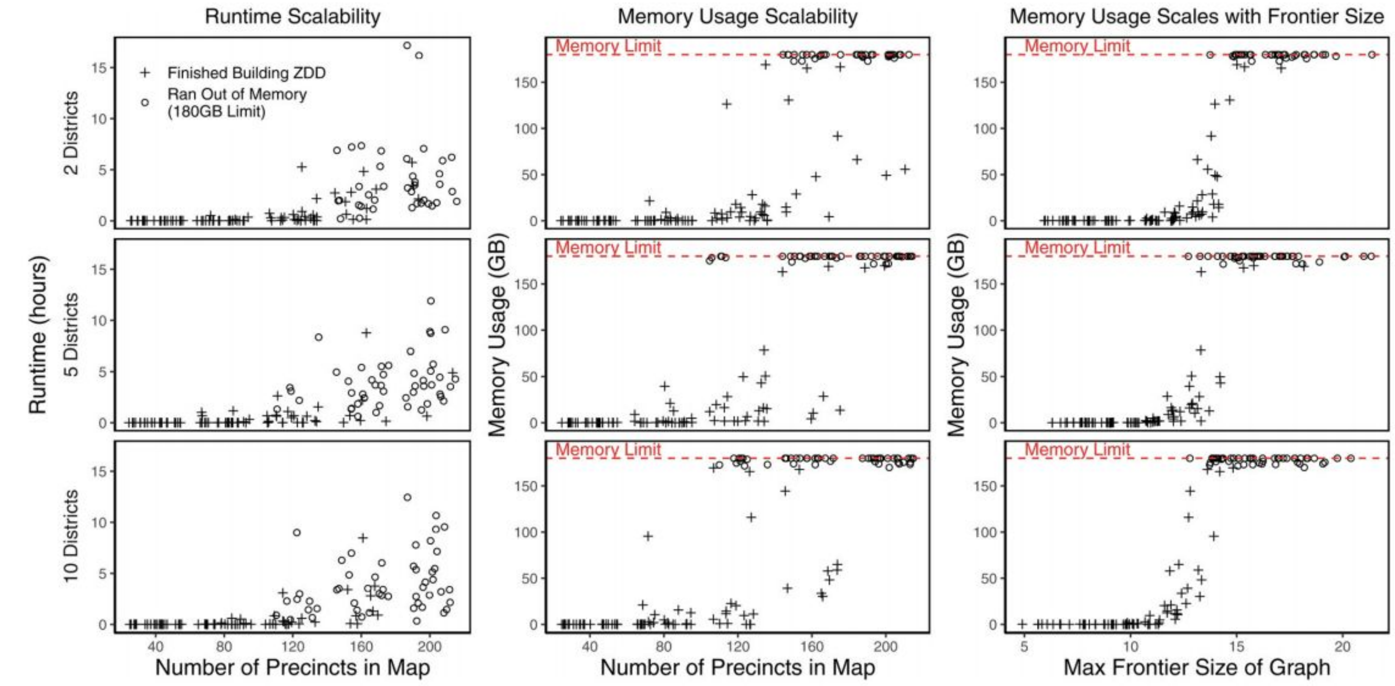


Figure 6. The scalability of the `enumpart` algorithm on subsets of the New Hampshire precinct map. This figure shows the runtime scalability of the `enumpart` algorithm for building the ZDD on random contiguous subsets of the New Hampshire precinct map. Crosses indicate maps where the ZDD was successfully built within the RAM limit of 180GB. In contrast, open circles represent maps where the algorithm ran out of memory. For the left and middle columns, the results are jittered horizontally with a width of 20 for the clarity of presentation. (The actual evaluation points on the horizontal axis are 40, 80, 120, 160, and 200.) The left column shows how total runtime increases with the number of units in the underlying map, while the center column shows how the total RAM usage increases with the number of units in the underlying map. Lastly, the right-hand column shows that memory usage is primarily a function of the maximum frontier size of the ZDD. We show results for 2-district partitions (top row), five-district partitions (middle row), and 10-district partitions (bottom row).

future work!

- We have a Julia implementation of weight-less enumpart working
  - this allows us to replicate Imai's workflow
  - compute number of *k*-partitions on grid graphs, Iowa, etc.
  - Needs speedups! Currently 6x6 -> 6 in 1 min, but we are only getting started!
  - For perspective, 9x9->9 is 9 orders of magnitude many solutions than 6x6->6
- write Julia version of weighted enumpart
  - see how fast we can make it
  - find the exact number of pop. balanced plans on IA, 10x10 → 10 parts.
- once **graphillion** is updated, see if it is faster!