A Recursive Combinatorial Description of cell-complex

- Author: Xie Yuheng

- Date: 2019-05-06
- Keywords: cell-complex, data structure.

Abstract

I provide a recursive combinatorial description of cell-complex,
which can be used to model higher order incidence relations in higher dimensional topological structures,
I hope it will serve as a stepstone for further formalization and experiments in algebraic topology.

Contains

Introduction

My description of cell-complex follows closely with the classical definition (such as in [3]),
excpet that I describe the incidence relation of cells more explicitly.

It is known that higher dimensional sphere recognition is undecidable[1].

But we should not conclude that combinatorial description cannot be given (such as in section 12 of [2]),
because "describable" (or "constructible") is weaker than "decidable".

And the construction of higher dimensional cell-complex by my method
is not limited by sphere recognition problem's undecidability.

- See section "Cell-complex (again, with comments)" for details.

Graph as an example (to help readers be familiar with the pseudo code)

In the following I use a javascript-like pseudo code to describe data structures.

- The postfix `_t` denotes type
- `id_t` is a serial number uniquely identify a vertex or an edge
- `dic_t <K, V>` is a dictionary (a finite map) from `K` to `V`

```
type id_t = number

class vertex_t {}

class edge_t {
  start: id_t
  end: id_t
}

class graph_t {
  vertex_dic: dic_t <id_t, vertex_t>
  edge_dic: dic_t <id_t, edge_t>
}
```

Cell-complex

`cell_complex_t` can be viewed as generalization of `graph_t` to higher dimension,

- Merge `vertex_dic` and `edge_dic` to `cell_dic`
- Add `dim` field in `id_t` to distinguish dimension

```
class id_t {
  dim: number
  ser: number
}

class cell_complex_t {
  cell_dic: dic_t <id_t, cell_t>
}

class cell_t {
  dom: spherical_t
  cod: cell_complex_t
  dic: dic_t <id_t, { id: id_t, cell: cell_t }>
}

class spherical_t extends cell_complex_t {
  spherical_evidence: spherical_evidence_t
}

class spherical_evidence_t {
  /**
   * [detail definition omitted]
   */
}
```

Cell-complex (again, with comments)

Comments in code block are written in `/** ... */`, while corresponding comments follows the code block.

```
class id_t {
  dim: number
  ser: number
}

class cell_complex_t {
  cell_dic: dic_t <id_t, cell_t>
}
```

To build a `cell_complex` we attach `cell`s to it iteratively,
while attaching a `cell` we also introduce a new `id` to uniquely identify the

`cell` within this `cell_complex`,

- where an `id` consist of a dimension and a serial number.

```
class cell_t {
  /**
   * `dom` -- domain
   * `cod` -- codomain
   */
  dom: spherical_t
  cod: cell_complex_t
  dic: dic_t <id_t, { id: id_t, cell: cell_t }>
}
```

When attaching a `cell` to a `cell_complex`, the `dom` must be a spherical cell-complex.
And the `cod` is the `n`-dimensional skeleton of the `cell_complex`, where `n` is the dimension of the `dom`.

Here the `dic` is a surjective map from id of `dom` to id to `cod`,
which serves as a record of how the `cell`s in `dom` are mapped to the `cell`s in `cod`.

- Here we can not simply use: `dic: dic_t <id_t, id_t>`
  we also need to record how the boundary of a cell `A` in `dom`
  is mapped to the boundary of the corresponding cell `B` in `cod`.
  we can record this extra information by another cell `C`, such that `C.dom == A.dom` & `C.cod == B.dom`.

- I found this only when trying to construct `vertex_figure` of `cell_complex`,
  without the extra information, it will be impossible to construct `vertex_figure`,
  and the construction of `vertex_figure` is important for checking whether a `cell_complex` is a `manifold`.

```
class spherical_t extends cell_complex_t {
  spherical_evidence: spherical_evidence_t
}
```

`spherical_t` is special `cell_complex_t`, it extends `cell_complex_t` by adding field `spherical_evidence`, which contains a homeomorphism between the `cell_complex` and a standard sphere (for example, boundary of n-simplex or n-cube).

- Homeomorphism between two cell-complexes is defined as isomorphism after subdivisions,
- and isomorphism between two cell-complexes is a generalization of isomorphism between two graphs.

It is known that higher dimensional sphere recognition is undecidable.

This means, for higher dimensional (d >= 5) sphere, we can not write a program to decide whether a cell-complex is homeomorphic to sphere.

- By "to decide" I mean to generate a proof, i.e. to construct the evidence of homeomorphism.

But for each specific cell-complex, it is always possible for one to provide the evidence of homeomorphism,

- i.e. not automatically generated by computer, but provided by human.

The definition of `cell_t` uses this evidence, but does not require a program to automatically generate `spherical_evidence` for all cell-complexes.

Thus the construction of higher dimensional cell-complex by my method is not limited by whether sphere recognition problem's decidable or not.

```
class spherical_evidence_t {
  /**
   * [detail definition omitted]
   */
}
```

Examples

`triangle` represented as javascript object

In the following example:

- `1:2` means an `id` of dimension `1`, serial number `2`
- `null` denotes `empty_cell`

The representation is designed to be readily serializable to JSON.

```
{ '0:0': null,
  '0:1': null,
  '0:2': null,
  '1:0':
   { dom: { '0:0': null, '0:1': null },
     cod: { '0:0': null, '0:1': null, '0:2': null },
     dic:
      { '0:0': { id: '0:0', cell: null },
        '0:1': { id: '0:1', cell: null } } },
  '1:1':
   { dom: { '0:0': null, '0:1': null },
     cod: { '0:0': null, '0:1': null, '0:2': null },
     dic:
      { '0:0': { id: '0:1', cell: null },
        '0:1': { id: '0:2', cell: null } } },
  '1:2':
   { dom: { '0:0': null, '0:1': null },
     cod: { '0:0': null, '0:1': null, '0:2': null },
     dic:
      { '0:0': { id: '0:2', cell: null },
        '0:1': { id: '0:0', cell: null } } } }
```

`triangle` defined as subclass of `cell_complex_t`

```
class triangle_t extends cell_complex_t {
  constructor () {
    let builder = new cell_complex_builder_t ()
    let [a, b, c] = builder.attach_points (3)
    let x = builder.attach_edge (a, b)
    let y = builder.attach_edge (b, c)
    let y = builder.attach_edge (c, a)
    super (builder)
  }
}
```

`torus` represented as javascript object

```
{ '0:0': null,
  '1:0':
   { dom: { '0:0': null, '0:1': null },
     cod: { '0:0': null },
     dic:
      { '0:0': { id: '0:0', cell: null },
        '0:1': { id: '0:0', cell: null } } },
```

```
    '1:1':
     { dom: { '0:0': null, '0:1': null },
       cod: { '0:0': null },
       dic:
        { '0:0': { id: '0:0', cell: null },
          '0:1': { id: '0:0', cell: null } } },
    '2:0':
     { dom:
        { '0:0': null,
          '0:1': null,
          '0:2': null,
          '0:3': null,
          '1:0':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null, '0:1': null, '0:2': null, '0:3': null },
             dic:
              { '0:0': { id: '0:0', cell: null },
                '0:1': { id: '0:1', cell: null } } },
          '1:1':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null, '0:1': null, '0:2': null, '0:3': null },
             dic:
              { '0:0': { id: '0:1', cell: null },
                '0:1': { id: '0:2', cell: null } } },
          '1:2':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null, '0:1': null, '0:2': null, '0:3': null },
             dic:
              { '0:0': { id: '0:2', cell: null },
                '0:1': { id: '0:3', cell: null } } },
          '1:3':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null, '0:1': null, '0:2': null, '0:3': null },
             dic:
              { '0:0': { id: '0:3', cell: null },
                '0:1': { id: '0:0', cell: null } } } },
       cod:
        { '0:0': null,
          '1:0':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null },
             dic:
              { '0:0': { id: '0:0', cell: null },
                '0:1': { id: '0:0', cell: null } } },
          '1:1':
           { dom: { '0:0': null, '0:1': null },
             cod: { '0:0': null },
             dic:
              { '0:0': { id: '0:0', cell: null },
```

```
                '0:1': { id: '0:0', cell: null } } } },
      dic:
       { '0:0': { id: '0:0', cell: null },
         '0:1': { id: '0:0', cell: null },
         '1:0':
          { id: '1:0',
            cell:
             { dom: { '0:0': null, '0:1': null },
               cod:
                { '0:0': null,
                  '1:0':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } },
                  '1:1':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } } },
               dic:
                { '0:0': { id: '0:0', cell: null },
                  '0:1': { id: '0:1', cell: null } } } },
         '0:2': { id: '0:0', cell: null },
         '1:1':
          { id: '1:1',
            cell:
             { dom: { '0:0': null, '0:1': null },
               cod:
                { '0:0': null,
                  '1:0':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } },
                  '1:1':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } } },
               dic:
                { '0:0': { id: '0:0', cell: null },
                  '0:1': { id: '0:1', cell: null } } } },
         '0:3': { id: '0:0', cell: null },
         '1:2':
```

```
          { id: '1:0',
            cell:
             { dom: { '0:0': null, '0:1': null },
               cod:
                { '0:0': null,
                  '1:0':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } },
                  '1:1':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } } },
               dic:
                { '0:0': { id: '0:1', cell: null },
                  '0:1': { id: '0:0', cell: null } } } },
         '1:3':
          { id: '1:1',
            cell:
             { dom: { '0:0': null, '0:1': null },
               cod:
                { '0:0': null,
                  '1:0':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } },
                  '1:1':
                   { dom: { '0:0': null, '0:1': null },
                     cod: { '0:0': null },
                     dic:
                      { '0:0': { id: '0:0', cell: null },
                        '0:1': { id: '0:0', cell: null } } } },
               dic:
                { '0:0': { id: '0:1', cell: null },
                  '0:1': { id: '0:0', cell: null } } } } } } }
```

`torus` defined as subclass of `cell_complex_t`

```
class torus_t extends cell_complex_t {
  constructor () {
```

```
    let builder = new cell_complex_builder_t ()
    let origin = builder.attach_point ()
    let toro = builder.attach_edge (origin, origin)
    let polo = builder.attach_edge (origin, origin)
    let surf = builder.attach_face ([
      toro,
      polo,
      toro.rev (),
      polo.rev (),
    ])
    super (builder)
  }
}
```

Remarks

Even for simple example like `torus`, the plain representation goes far beyond the cognitive complexity I can endure.

And indeed, instead of using the plain object representation, the intended usage is to abstract over the basic data structures, and, layer by layer, design higher level interface functions.

- This is how people control the cognitive complexity in computer science in general.

The `cell_t` is recursively defined in the same way for all dimensions, but each dimension is special.
And, for example, interface functions such as `attach_point`, `attach_face`, `attach_body` can be designed for each specific dimension.

More example cell-complexes can be found at the main project page.

- Further documentation about programming interface is work in progress.

Note about incidence matrix

`dic_t` can be viewed as sparse matrix.

The recursive definition of `cell_t` means that, instead of incidence matrix, we need nested higher order incidence matrix to describe cell-complex.

For example, for 1-dimensional edges, we can use incidence matrix (like in graph theory),

while for 2-dimensional faces, to represent the incidence relation,
we need a matrix valued matrix, where the inner matrix encode the orientation of the incidence relation.

- For a directed graph, we can simply use `+1` or `-1` to encode the orientation,
  the incidence relation can be represented by a `+1, -1` valued matrix.

- while for a 2-dimensional face, we need a matrix to encode the orientation,
  the incidence relation can be represented by matrix valued matrix.

- and for a 3-dimensional body, we need a matrix of matrix to encode the orientation,
  the incidence relation can be represented by *matrix valued matrix* valued matrix.

and so on and so forth ...

- Note that, the type (or shape) of inner matrix depends on the shape of cell boundary,
  thus these nested matrixes are not exactly higher order tensors.

Note about space complexity

Due to the recursive construction, the space increases exponentially with the dimension.

If the dimension is bounded by `d`, the space complexity is `O(n^d)`, where `n` is the number of `d` dimension cells.

Future works

Based on the basic construction of cell-complex, I plan to:

- Generalize the relation between 2-dimensional cell-complex
  and the presentation theory of groupoid to higher dimension.
- Provide more online interactive tools to help people study cell-complexes and algebraic topology,
  - The library is developed for javascript with this aim in mind.

Appendixes

- A Substitution Model for Class Definition
  - Further clarify the use of class definitions in this paper for people with less programming experiences.
  - Also summarize the difference between "describable" and "decidable".

References

[1] Unrecognizability of manifolds

- by A.V. Chernavsky, V.P. Leksine.

[2] Topological methods, in: "Handbook of Combinatorics"

- by Anders Björner, (R. Graham, M. Grötschel, and L. Lovász, eds.)

[3] The Topology of CW Complexes

- by Albert T. Lundell, Stephen Weingram