Lecture 12
Combinatorial Planning in High Dimensions

Instructor: Jingjin Yu
Outline

Review of combinatorial motion planning in the plane
Completeness and the halting problem
Cell decomposition in high dimensions
Complexity of motion planning
Last time, we covered several **combinatorial motion planning** algorithms in the plane

- Vertical cell decomposition
- Shortest-path roadmaps
- Maximum clearance roadmaps

What do these have in common?
- Each provides a (**combinatorial**) partitioning of the environment
- Which makes these algorithms **complete**
Completeness of Algorithms

Definition: an algorithm (in the sense of Turing machine) is complete if it always returns in finite time a positive or negative answer

Behavior of a complete algorithm

⇒ If the problem instance is solvable, it returns the solution
⇒ If the problem instance is not solvable, it must return “no solution”

Are all algorithms complete?

Note that this is different from the question of whether all problems are solvable with algorithms

⇒ How did we prove that not all problems can be solved with an algorithm?

|       | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ | $M_6$ | ...
|-------|-------|-------|-------|-------|-------|-------|-------|
| $P_1$ | Yes   | No    | Yes   | Yes   | No    | No    | ...
| $P_2$ | No    | No    | Yes   | No    | Yes   | No    | ...
| $P_3$ | No    | Yes   | Yes   | Yes   | No    | Yes   | ...
| $P_4$ | Yes   | No    | No    | Yes   | No    | Yes   | ...
| ...   |       |       |       |       |       |       | ...
| $P_{unsolvable}$ | No | Yes | No | No | ... | ... | ... |
The Halting Problem

The **halting problem**: is it possible to determine whether a program will stop?

⇒ This problem is undecidable (using another algorithm)!
⇒ That is, a computer (Turing machine) cannot really tell

**Sketch of proof (via contradiction)**

⇒ Suppose an algorithm $H_{TM}$ can decide whether an algorithm $X$ will halt
⇒ Construct a new algorithm $A_{TM}$

\[
A_{TM}: \text{on input algorithm } X \\
\text{if } H_{TM}(X) == \text{true then loop forever; } \\
\text{else return true;}
\]

What happens if we run $A_{TM}$ on input $A_{TM}$?

⇒ If $A_{TM}$ halts, then $H_{TM}(A_{TM}) == \text{true, } A_{TM}$ should loop forever
⇒ If $A_{TM}$ does not halt, then $H_{TM}(A_{TM}) == \text{false, } A_{TM}$ should halt
⇒ Both are contradictions $\rightarrow H_{TM}$ cannot exist!
Implications of the Halting Problem

So, are all algorithms complete?

⇒ No!
⇒ Proof sketch
  ⇒ There exist algorithms which we cannot tell whether they will stop
  ⇒ Such algorithms may run forever and there is nothing we can do
  ⇒ Such algorithms/programs are not complete
⇒ In practice, this can be bad
  ⇒ E.g., real time systems
  ⇒ Solution: do not use full Turing machine

Combinatorial algorithms are complete

⇒ This is because every single point in $C_{free}$ is covered
⇒ This is a big deal – a piece of mind
⇒ Motivates the development of combinatorial methods for higher dimensions
A classical “grand challenge” in robotics was the “Piano Mover’s Problem”

- Using autonomous robots, how to move a piano around without hitting other objects?

It was not immediately clear whether such problems are solvable

- Addressed by Schwartz and Sharir in a series of classical papers (‘83)
- The problem is shown to be PSPACE-hard (Reif ‘79)
- But solvable with Collins’ cylindrical algebraic decomposition (‘75)
- Canny provided a “faster” algorithm a few years later (‘87)

We will look at these algorithms briefly

A Formal Problem Definition

Formulation 4.1 (The Piano Mover’s Problem)

1. A world $\mathcal{W}$ in which either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.

2. A semi-algebraic obstacle region $\mathcal{O} \subset \mathcal{W}$ in the world.

3. A semi-algebraic robot is defined in $\mathcal{W}$. It may be a rigid robot $\mathcal{A}$ or a collection of $m$ links, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$.

4. The configuration space $\mathcal{C}$ determined by specifying the set of all possible transformations that may be applied to the robot. From this, $\mathcal{C}_{\text{obs}}$ and $\mathcal{C}_{\text{free}}$ are derived.

5. A configuration, $q_I \in \mathcal{C}_{\text{free}}$ designated as the initial configuration.

6. A configuration $q_G \in \mathcal{C}_{\text{free}}$ designated as the goal configuration. The initial and goal configurations together are often called a query pair (or query) and designated as $(q_I, q_G)$.

7. A complete algorithm must compute a (continuous) path, $\tau : [0, 1] \rightarrow \mathcal{C}_{\text{free}}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, or correctly report that such a path does not exist.
Extending VCD to higher dimensions (say $n$ dimensions)

- Pick a dimension $x_i$
- Sweep the configuration space using a $(n - 1)$-dimensional plane orthogonal to $x_i$ (this is unique)
  - Similar to 2D case, stop only when “critical connectivity events” happen
  - That is, with the appearance/disappearance/change of $C_{obs}$
  - Every time a critical change happens, a new $(n - 1)$-dimensional slice is produced
- Repeat the process to each of the $(n - 1)$-dimensional slices
- The sweeping yields convex $n$-cells, $(n - 1)$-cells, and so on...
  - An $n$-cell is a $n$-dimensional polytope sandwiched between $(n - 1)$-dimensional slices
  - E.g., in 2D, 2-cells and 1-cells
Vertical Cell Decomposition in 3D

For three dimensions

⇒ First do plane sweep
⇒ Then line sweep

Example
Issues with Vertical Cell Decomposition

VCD requires $C_{\text{obs}}$ be polytopes (high dimensional “polygons”)

$\Rightarrow$ This is the case for translating rigid bodies among obstacles that are polytopes

$\Rightarrow$ However, configuration spaces are more like this

Need more powerful methods!
Computational Algebraic Geometry

Recall that we have defined configuration space using semi-algebraic sets

\[ f_1 = x^2 + y^2 < 1 \]
\[ f_2 = (x - \frac{1}{2})^2 + (y - \frac{1}{2})^2 > 0.01 \]
\[ f_3 = (x + \frac{1}{2})^2 + (y - \frac{1}{2})^2 > 0.01 \]
\[ f_4 = \frac{x^2}{2} + (y + \frac{1}{2})^2 > 0.01 \]

Such sets can describe the configuration space of complex motion planning problems

Usually this is done over rational polynomials \( \mathbb{Q}[x_1, \ldots, x_n] \)

\[ \Rightarrow \text{Computing roots for real polynomials is tricky (precision wise)} \]
\[ \Rightarrow \text{Note that this doesn’t mean no irrational numbers will appear!} \]
\[ \Rightarrow \text{E.g., } x^2 - 2 \geq 0. \text{ But all these can be represented compactly} \]
\[ \Rightarrow \text{Polynomials with rational coefficient are nice: the projection of a semi-algebraic set from dimension } n \text{ to dimension } n - 1 \text{ is a semi-algebraic set!} \]
Cylindrical Algebraic Decomposition

Cylindrical algebraic decomposition is similar to VCD in spirit

⇒ Start with a set of $n$ dimensional polynomials $\mathcal{F}_n$
⇒ Projects from $n$ to $n-1$ dimensions
  ⇒ Figure out where the polynomials fold or intersect
⇒ The information is contained in the projection $\mathcal{F}_{n-1}$
⇒ Once we get to $\mathcal{F}_1$, this is a one dimensional polynomial
⇒ Partitions $\mathbb{R}$ into 0-cells and 1-cells
⇒ Then reverse the process
  ⇒ From $\mathbb{R}^1 \to \mathbb{R}^2$, 0-cells expand to 0-cells and 1-cells
  ⇒ 1-cells expand to 2-cells
⇒ Going from $\mathbb{R}^{n-1} \to \mathbb{R}^n$, the full decomposition
⇒ 2D example:
Cylindrical algebraic decomposition is slow!

⇒ It is doubly exponential with respect to the number of dimensions
⇒ That is, the running time contains a term $a^{bn}$ for $a, b > 1$
⇒ John Canny invented a method to “speed” it up
⇒ The basic idea is to project directed from $\mathbb{R}^n$ to $\mathbb{R}^2$ many times
⇒ The technique drops the computation time to singly exponential ($a^n$)
⇒ Still not very practical but theoretically much better

This was John Canny’s Ph.D. Thesis, which won ACM best thesis award

But he may as well be more famous for his MS thesis: Canny’s edge detector
Complexity of Motion Planning (I)

Complexity classes: P, NP, PSPACE, EXPTIME

- For a problem with an input size $n$
  - $P$: the problem is solvable in time $n^k$ for some $k < \infty$
  - $NP$: the solution to the problem can be verified in time $n^k$ for some $k < \infty$
  - $PSPACE$: the problem can be solved by a Turing machine using only $n^k$ space for some $k < \infty$
  - $EXPTIME$: the problem can be solved in time $2^{n^k}$ for some $k < \infty$
- We know $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$
- And $P \neq EXPTIME$

NP-hard and PSPACE-hard

- A (decision) problem is NP-hard if it is “harder” to solve than any other problem in NP
- If it is also in NP, then it is NP-complete
- Similarly, we can define PSPACE-hardness and PSPACE-completeness
Complexity of Motion Planning

The Piano Movers Problem was shown to be PSPACE-HARD (Reif, ‘79)

Note that this does not mean the problem is in PSPACE!

Every problem outside of PSPACE would also be PSPACE-hard by definition

This is a “lower bound” on the difficulty of the problem

Canny’s algorithm shows that the problem is in PSPACE

This is an “upper bound” on the difficulty of the problem

Together the problem is then PSPACE-complete

Implications

PSPACE-hard is believed to be (much) harder than NP-hard

Need alternative solution methods

Perhaps we have to sacrifice completeness

This leads to sampling based algorithms, which has a weaker notion of completeness – probabilistic completeness