Lecture 17
Multi-Robot Path Planning (2)
Outline

NP-Hardness of $\text{MPP}_r$
- NP and NP-hardness
- Reduction from 3-SAT

Algorithms for multi-robot path and motion planning
- Graph search based algorithm for $\text{MPP}_p$
- Integer linear programming models for $\text{MPP}_r$
NP and NP-Hardness

Note that we are classifying **problems** here! In particular, we are NOT classifying **algorithms** (a common mistake).

A problem is in the class **non-deterministic polynomial time (NP)** if
- It can be solved by a **non-deterministic Turing machine** in polynomial time
- Equivalently, a given solution can be verified in polynomial time
- E.g., graph search
- E.g., given a graph $G$, find a Hamiltonian cycle
  - To see that it is in NP, given a cycle, verifying it is part of $G$ is doable in polynomial time

A problem $P_1$ is **NP-hard** if
- Solving it is harder than solving any other problems in NP
- I.e., any problem $P_2 \in \text{NP}$ can be solved in polynomial time via solving $P_1$
- Note that a problem is NP-hard does not require it is in NP
- A problem that is NP-hard and also in NP is NP-complete
- This implies that all NP-complete problems are in a sense “equal” in hardness
Some Classical NP-Complete Problems

**Boolean satisfiability (SAT):** first problem proven to be NP-hard

- $n$ binary variables $x_1, \ldots, x_n$
- A literal $y$ of a variable $x$ is $x$ or $\neg x$, total $2n$ of these
- $m$ disjunctive clauses of literals, i.e. $c_j = y_{j_1} \lor \cdots \lor y_{j_k}$
- Question: are there values for $x_i, \ldots, x_n$ so that $c_1 \land \cdots \land c_m = 1$?
- Shown to be NP-complete (Cook-Levin theorem)
  - SAT is in NP because checking an answer is doable in polynomial time
  - NP-hard via direct reduction from a generic nondeterministic Turing machine

**3SAT:** SAT with each clause containing up to 3 literals

- NP-hard via the reduction from SAT
- Reduction is how NP-hardness is proven in general

**Vertex cover:** $G = (V, E)$, is there a set of $K$ vertices that covers $V$?
- Reduction from 3SAT

Numerous others: Hamiltonian cycle, Traveling Salesperson Problem (TSP), Set Cover, Knapsack, ...
NP-Hardness of Makespan Optimal MPP$_r$

Min Makespan MPP$_r$ is NP-hard

$$\text{3SAT} \quad (x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor x_2 \lor \neg x_4) \land (\neg x_2 \lor x_3 \lor x_4)$$

$n = 4$ variables
$m = 3$ clauses

Min Makespan MPP
NP-Hardness of Distance Optimal MPP

NP-hardness of distance optimal MPP is slightly more tricky...

**Theorem.** MPP is NP-hard when optimizing min makespan, min total time, min max distance, and min total distance.
Implications of MPP Intractability

A problem being NP-hard means likely no polynomial time algorithm exists for solving it exactly

⇒ More precisely, unless P = NP (a million dollar question)

Implications:

⇒ Practitioners should not try to find polynomial time algorithm
⇒ Aiming for solving the problem approximately
⇒ Or aiming for solving easier cases of the problem quickly
⇒ This is what we will try with MPP

Algorithmic solutions for MPP fall into two flavors

⇒ Discrete search based algorithms
⇒ Integer programming solvers
Discrete Search Algorithms

General methods

- **Coupled** search – treat all robots as a “single” robot
- **Decoupled** search – treat robots as individual ones as much as possible
- Recall the basic structure of search algorithms (still applies!)

```plaintext
input: G = (V, E), x_i, x_G
AddToQueue(x_i, Queue); // Add x_i to a queue of nodes to be expanded
while(!IsEmpty(Queue))
    x ← Front(Queue); // Retrieve the front of the queue
    if(x.expanded == true) continue; // Do not expand a node twice
    x.expanded = true; // Mark x as expanded
    if(x == x_G) return solution; // Return if goal is reached
    for each neighbor n_i of x // Add all neighbors of x to the queue
        if(n_i.expanded == false) AddToQueue(n_i, Queue)
return failure;
```
Coupled Search

Key: treat all robots as a whole

⇒ For a single robot, # neighbors?
  ⇒ Up to 4 neighbors

⇒ What about multi-robot case?
  ⇒ Up to 5 neighbors per robot, including staying put

⇒ The search works in a straightforward way
  ⇒ For three robots, a neighboring node may be (east, north, stay)

⇒ But, huge branching factor!
  ⇒ For $n$ robots, $5^n$ “neighbors” in the graph
  ⇒ For 3 robots, 125 neighbors per step
  ⇒ Optimal, complete, but impractical (why exactly?)
Coupled Search – Why Impractical?

How large can a priority queue be?

⇒ For a single robot, no more than $|V|$.
⇒ For $n$ robots, $|V|^n$.
⇒ Well, not exactly, a bit smaller, why?
⇒ But close enough.
⇒ Suppose $|V| = 10^3$, $n = 10$.
⇒ $|V|^n = 10^{30}$!
⇒ We cannot hope to even store the queue on hard disk.
⇒ So search will be extremely slow!
Decoupled Search

Key idea: treat robots as individual ones as much as possible

⇒ To start, plan optimal paths for individual robots
  ⇒ This reduces branching factor: $5^n \Rightarrow 4n$
⇒ Then, simulate the “execution” of the paths
⇒ When there are conflicts, push all choices onto the priority queue
⇒ Then continue the “execution”
⇒ Initial paths may get updated/changed
⇒ In the example
  ⇒ One queue node corresponding to robot 1 takes the junction first
  ⇒ One queue node corresponding to robot 2 takes the junction first
  ⇒ Both will add an additional (makespan) cost of 1
⇒ A rough sketch: practical implementations require lots of care-taking
Handling Different Objectives

Different objectives cause the queue to be sorted differently

⇒ Total distance
  ⇒ Initially all choose left path
  ⇒ Then 1-4 have conflict at $t = 1$, generating 4 new nodes (robot $i$ goes first)
  ⇒ Then at $t = 2$, suppose we pick the node letting 1 go first, three new nodes are created
  ⇒ These three new nodes can be inserted into the front of the queue using a secondary heuristic
  ⇒ After one more iteration, 2 new nodes are generated
  ⇒ Then one last iteration resolves all conflicts
  ⇒ The total distance remains the same for all nodes, which is $4x + 8$

⇒ Total time
  ⇒ Initial node cost is $4x + 8$
  ⇒ Here, at $t = 1$, 4 new nodes, cost is now $4x + 11$ for all
  ⇒ At $t = 2$, if 4 goes through the right, cost is $4x + 13$, otherwise, $4x + 14$
Strengths and Weakness of Discrete Search

Strengths of discrete search solutions

- When it works, the algorithm generally runs rather fast
  - Because the overall algorithm is relatively simple due to its discrete nature
- Capable of solving large (sparse) problems
- Generally straightforward to implement and tweak

Weaknesses

- As the interactions among the robots grow, performance degrades quickly
- As such, not suitable for solving very dense problems
- Not suitable for handling MPP\textsubscript{T} as the number of possible rotations can be very large; huge branching factor
  - For 16-puzzle, >1000 possible cycles
  - Each cycle has two directions
  - Enumerating becomes impossible
An Integer Programming Based Solver for MPP$_r$

Transform MPP into multiflow over time steps

Constraints

Each edge is a binary variable

$$
\sum_{1 \leq i \leq n} (x_{uv,t,t+1}^i + x_{vv,t,t+1}^i + x_{wv,t,t+1}^i) \leq 1
$$

$$
\sum_{1 \leq i \leq n} (x_{uv,t,t+1}^i + x_{vu,t,t+1}^i) \leq 1
$$
An Example
The ILP-base algorithm can require big models

- $20 \times 15$ grid, 20 steps, 20 robots $\rightarrow \sim 1$ million variables
- We can use a divide-and-conquer like heuristic through **splitting over time**
- The algorithm is no longer complete and may yield sub-optimal solutions

**Additional Splitting Heuristic**
The Approach Can Solve Some Tough Problem

A 7-step min makespan plan

10^{25} states
> 10^4 branching factor
Some Examples in 2D Continuous Domain

No static obstacles, 75 robots
1.7 seconds to compute, 1.6-optimal

Random obstacles, 50 robots
4.0 seconds to compute, 1.9-optimal