Contents

- Multi-Agent Path Finding
- Goal Oriented Planning
Multi-Agent Path Finding (MAPF)

Input
- A graph with $n$ vertices, usually a grid with obstacles
- A set of $k$ agents each with a start and goal vertex

The Problem
- An agent can move or wait in each time step
- The task is to find a path for each agent such that
  - paths do not conflict (two agents at the same vertex)
  - optimize makespan or cost of all paths
- The problem is solved by a centralized solver offline
MAPF example
MAPF Motivation

- Robotics
- Warehouse logistics
- Video Games (mostly Strategy and Role-playing games)
Sliding puzzle – a special case for MAPF
Plan Existence

Some trivial conditions for plan existence

- No two agents have the same starting node
- No two agents have the same goal node
- The number of agents is less than number of nodes
Plan Existence

Some trivial conditions for plan existence

- No two agents have the same starting node
- No two agents have the same goal node
- The number of agents is less than number of nodes

How about this?
Plan Existence

Some trivial conditions for plan existence

- No two agents have the same starting node
- No two agents have the same goal node
- The number of agents is less than number of nodes

How about this?

No-swap constraint is often relaxed

- Agents may swap positions – use the same edge at the same time
No-swap constraint is often relaxed

- Agents may swap positions – use the same edge at the same time
No-Train Constraint

- Agents may (or may not) form a train when moving
  - Agent can approach a node that is occupied now but will be free in the next step.
  - Has effect on makespan and cost but not solvability
Generalizing No-Train and No-Swap

k-robustness

- An agent can visit a node if that node has not been visited in the last recent $k$ steps.
- 1-robustness covers no-swap and no-train

Why k-robustness?
Generalizing No-Train and No-Swap

k-robustness

- An agent can visit a node if that node has not been visited in the last recent \( k \) steps.
- 1-robustness covers no-swap and no-train

Why k-robustness?

- Safety reasons: if an agent is delayed a new agent may collide with him when executing the plan.
Constraints and Objectives

Other possible constraints

- No path has a cycle
- Waiting not allowed
- A node may be visited on \( k \) times
- ...

How to measure plan quality?

- Makespan – number of steps for each agent to get to their destination
- Sum of Costs (SoC) – sum of lengths of individual agents plans (plus cost of waiting if given)
Complexity

Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
- Single Agent – e.g., Dijkstra's algorithm
- Joint goal nodes, i.e., it does not matter which agent reaches which goal – can be reduced to min-cost-flow problem.
- Sub-optimal MAPF with at least two unoccupied nodes – several polynomial algorithms exist such as "push and swap" (Luna and Bekris 2011) or "push and rotate" (Wilde et.al. 2014).
Complexity

- Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
Complexity

- Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
- Single Agent
Complexity

- Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
- Single Agent – e.g., Dijkstra’s algorithm
Complexity

- Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
- Single Agent – e.g., Dijkstra’s algorithm
- Joint goal nodes, i.e., it does not matter which agent reaches which goal – can be reduced to min-cost-flow problem.
Complexity

- Optimal makespan or SoC MAPF is NP-Hard

Easy (Polynomial) cases:
- Single Agent – e.g., Dijkstra’s algorithm
- Joint goal nodes, i.e., it does not matter which agent reaches which goal – can be reduced to min-cost-flow problem.
- Sub-optimal MAPF with at least two unoccupied nodes – several polynomial algorithms exist such as ”push and swap” (Luna and Bekris 2011) or ”push and rotate” (Wilde et.al. 2014).
How to solve MAPF?

- Encode in PDDL and use a generic planner. Too slow, does not scale well for large problems.
- Polynomial sub-optimal rule-based algorithms like push&swap, push&rotate, bibox, ... scale better but the solution quality is usually rather bad.
- Search based techniques: State space search with A* and some heuristics. Conflict based search – kinda sorta plan-space-search work well for many "practical" problems.
- Reduction based techniques: SAT/MaxSat/ILP/CSP/ASP based MAPF work best for small sized but complex problem instances.
How to solve MAPF?

- Encode in PDDL and use a generic planner
  - Too slow, does not scale well for large problems

- Polynomial Sub-optimal rule-based algorithms
  - Like push&swap, push&rotate, bibox, ...
  - Scale better but the solution quality is usually rather bad

- Search based techniques
  - State space search with A* and some heuristics.
  - Conflict based search – kinda sorta plan-space-search
    - work well for many "practical" problems

- Reduction based techniques
  - SAT/MaxSat/ILP/CSP/ASP based MAPF
    - work best for small sized but complex problem instances
How to solve MAPF?

- Encode in PDDL and use a generic planner
  - Too slow, does not scale well for large problems
- Polynomial Sub-optimal rule-based algorithms
  - Like push\&swap, push\&rotate, bibox, ...
  - Scale better but the solution quality is usually rather bad
How to solve MAPF?

- Encode in PDDL and use a generic planner
  - Too slow, does not scale well for large problems
- Polynomial Sub-optimal rule-based algorithms
  - Like push&swap, push&rotate, bibox, ...
  - Scale better but the solution quality is usually rather bad
- Search based techniques
  - State space search with A* and some heuristics.
  - Conflict based search – kinda sorta plan-space-search
  - work well for many “practical” problems
How to solve MAPF?

- Encode in PDDL and use a generic planner
  - Too slow, does not scale well for large problems
- Polynomial Sub-optimal rule-based algorithms
  - Like push&swap, push&rotate, bibox, ...
  - Scale better but the solution quality is usually rather bad
- Search based techniques
  - State space search with A* and some heuristics.
  - Conflict based search – kinda sorta plan-space-search
  - Work well for many “practical” problems
- Reduction based techniques
  - SAT/MaxSat/ILP/CSP/ASP based MAPF
  - Work best for small sized but complex problem instances
State Space Search for MAPF

- What is a state?
  - The position of each agent

- What is a transition step?
  - A move or wait for each agent
  - Branching factor \((d + 1)^k\)
  - It is easy to think of heuristics, but still huge branching factor
  - Consider 15 agents on a grid: \(15^5 = 30, 517, 578, 125\)
  - Just to calculate the successors for A* and evaluate the heuristic would take forever

\[1\] where \(d\) the vertex degree and \(k\) number of agents.
Cooperative A* – Silver 2005

- The algorithm tries to address the huge branching factor
- Basic Idea
  - Plan for each agent separately
  - Avoid collisions by waiting until another agent moves out of the way

Implementation
- Reservation table – a data structure to hold the location of each agent in each time step
- Find a path for the first agent and save it in the reservation table
- Find a path for the \( k \)-th agent by taking into consideration the reservation table with the paths of agent 1, 2, ..., \( k-1 \).
The algorithm tries to address the huge branching factor

**Basic Idea**
- Plan for each agent separately
- Avoid collisions by waiting until another agent moves out of the way

**Implementation**
- Reservation table – a data structure to hold the location of each agent in each time step
- Find a path for the first agent and save it in the reservation table
- Find a path for the $k$-th agent by taking into consideration the reservation table with the paths of agent $1, 2, ..., k - 1$
Cooperative A* – Properties

- **Pros**
  - It is sound (correct result)
  - It is polynomial in grid size and time

- **Cons**
  - It is not optimal
  - It is not complete – may not find a solution when it exists
Another way to address the branching factor

- At each transition only one agent is considered

- $s_j^k$ is the $j$-th step of $k$-th agent, the decisions are $s_1^1, s_2^1, s_3^1, \ldots, s_1^2, s_2^2, s_3^2, \ldots$

Pros

- Branching factor is reduced (5 for a grid)
- With a good heuristic it can solve the problem
- Complete and optimal if used in A* with an optimal heuristic

Cons

- Solution depth increases a lot (by a factor of number of agents)
- More nodes must be expanded (more memory requirement)
Find an optimal path for each agent ignoring the other agents

**While** (there is a group of agents that has a collision) **do**:
merge the agents into one group and solve this group optimally

In the example red and green get merged, yellow is unaffected.
Find an optimal path for each agent ignoring the other agents

**While** (there is a group of agents that has a collision) do:

*try to replace the path of one agent with a same cost solution, if not possible then* merge the agents into one group and solve this group optimally
M* – Wagner and Choset 2011

Find an optimal path for each agent ignoring the other agents
Start search, but only expand nodes on optimal paths of all agents
If conflict occurs then backtrack and expand all nodes
Apply M* recursively
Conflict Based Search (CBS)

Basic Idea

- Plan for each agent individually
- Validate Plans
- If agent A and B collide then
  - either constraint A to avoid the collision and replan A
  - or constraint B to avoid the collision and replan B

Properties

- resembles plan space planning
- works well for bottlenecks with many agents
SAT Based MAPF

Variables: $X_{k,j}^i$ – Agent $k$ is at node $j$ at step $i$

Clauses:

- For each agent $k$ and time step $i$:
- $\bigwedge_{j \neq j'} (\neg X_{k,j}^i \vee \neg X_{k,j'}^i)$ – agent $k$ is at most at one node in step $i$
- $(X_{k,1}^i \lor X_{k,2}^i \lor \ldots \lor X_{k,N}^i)$ – agent $k$ is at least at one node in step $i$

What about no-swap and no-train constraints?
SAT Based MAPF

Variables: $X_{k,j}^i$ – Agent $k$ is at node $j$ at step $i$
Clauses:

- For each agent $k$ and time step $i$:
  - $\bigwedge_{j \neq j'} (\neg X_{k,j}^i \lor \neg X_{k,j'}^i)$ – agent $k$ is at most at one node in step $i$
  - $(X_{k,1}^i \lor X_{k,2}^i \lor \ldots \lor X_{k,N}^i)$ – agent $k$ is at least at one node in step $i$
- For each node $j$ and time step $i$:
  - $\bigwedge_{k \neq k'} (\neg X_{k,j}^i \lor \neg X_{k',j}^i)$ – at most one agent at one node $j$ in step $i$
SAT Based MAPF

Variables: $X_{k,j}^i$ – Agent $k$ is at node $j$ at step $i$

Clauses:

- For each agent $k$ and time step $i$:
  - $\bigwedge_{j \neq j'} (\neg X_{k,j}^i \lor \neg X_{k,j'}^i)$ – agent $k$ is at most at one node in step $i$
  - $(X_{k,1}^i \lor X_{k,2}^i \lor \ldots \lor X_{k,N}^i)$ – agent $k$ is at least at one node in step $i$

- For each node $j$ and time step $i$:
  - $\bigwedge_{k \neq k'} (\neg X_{k,j}^i \lor \neg X_{k',j}^i)$ – at most one agent at one node $j$ in step $i$

- For each agent $k$ and each time step $i$:
  - $(X_{k,j}^{i+1} \implies X_{k,j}^i \lor \bigvee_{j' \in N(j)} X_{k,j'}^i)$ – if agent $k$ is at node $j$ in time $i + 1$ then it was already there last time or and neighboring node $j' \in N(j)$
SAT Based MAPF

Variables: $X^i_{k,j}$ – Agent $k$ is at node $j$ at step $i$

Clauses:

- For each agent $k$ and time step $i$:
  - $\bigwedge_{j \neq j'} (\neg X^i_{k,j} \lor \neg X^i_{k,j'})$ – agent $k$ is at most at one node in step $i$
  - $(X^i_{k,1} \lor X^i_{k,2} \lor \ldots \lor X^i_{k,N})$ – agent $k$ is at least at one node in step $i$

- For each node $j$ and time step $i$:
  - $\bigwedge_{k \neq k'} (\neg X^i_{k,j} \lor \neg X^i_{k',j})$ – at most one agent at one node $j$ in step $i$

- For each agent $k$ and each time step $i$:
  - $(X^{i+1}_{k,j} \implies X^i_{k,j} \lor \bigvee_{j' \in N(j)} X^i_{k,j'})$ – if agent $k$ is at node $j$ in time $i + 1$ then it was already there last time or and neighboring node $j \in N(j)$

- What about no-swap and no-train constraints?
SAT Based MAPF

Variables: $X_{k,j}^i$ – Agent $k$ is at node $j$ at step $i$

No-swap/no-train Clauses:

- For each agent $k$ and each pair of nodes $j, j' \in N(j)$ and each time $i$:
  
  - $(X_{k,j}^i \land X_{k,j'}^{i+1} \implies \bigwedge_{k'} X_{k',j'}^i)$ – if agent $k$ moves from $j$ to $j'$ then $j'$ has to be free (there is no agent there) before the move.
SAT Based MAPF

Variables: $X^i_{k,j}$ – Agent $k$ is at node $j$ at step $i$

No-swap/no-train Clauses:

- For each agent $k$ and each pair of nodes $j, j' \in N(j)$ and each time $i$:
  $$(X^i_{k,j} \land X^{i+1}_{k,j'} \implies \bigwedge_{k'} X^i_{k',j'})$$ – if agent $k$ moves from $j$ to $j'$ then $j'$ has to be free (there is no agent there) before the move.

- The ”$\bigwedge$” after implication makes the formula into $K$ clauses. Total clauses of this kind: $4|A|^2 \cdot |T| \cdot |V|$
Planning for A.I. in Video Games

- **Challenges**
  - We need real-time planning (very fast reactions)
  - continuous, nondeterministic, partially observable environments
  - There are usually other (unpredictable) agents in the world (like the human player)
  - How to model properly (how to choose the goals)?

- **What is used instead**
  - Reactive techniques (Finite state machines, Behavior Trees)
  - Machine learning (very rare, Black and White, Creatures series)
Goal Oriented Action Planning – GOAP

A.I. Planning for Computer Games (FPS/RPG kind) used in games like

- F.E.A.R. (X360/PS3/PC) - Monolith Productions/VU Games, 2005
- Ghostbusters (Wii) - Red Fly Studio, 2008
- Silent Hill: Homecoming (X360/PS3) - Double Helix Games/Konami, 2008
- Fallout 3 (X360/PS3/PC) - Bethesda Softworks, 2008
- Empire: Total War (PC) - Creative Assembly/SEGA, 2009
- Demigod (PC) - Gas Powered Games/Stardock, 2009
- LMNO (working title) (X360/PS3) - Electronic Arts
- Just Cause 2 (PC/X360/PS3) - Avalanche Studios/Eidos Interactive, 2010
- Deus Ex: Human Revolution (PC/X360/PS3) - Eidos Interactive, 2011
GOAP

- Introduced by Jeff Orkin in the game F.E.A.R.
  [http://alumni.media.mit.edu/~jorkin/goap.html](http://alumni.media.mit.edu/~jorkin/goap.html)
- Very positive reception by players and critics
- Inspired by STRIPS planning, similarities are:
  - Actions with preconditions, effects and cost
  - Problems are specified by their goal conditions
  - Planning is done backwards from goal towards initial state
- Differences to STRIPS
  - additional procedural preconditions – to check complex conditions, i.e., is there a path to safety (a precondition for a flee action)
  - additional procedural effects – to execute complex effects, i.e., if the fee action is selected a destination and path has to be selected
**GOAP**

- Overall A.I. System
  - the designers specify the possible actions for bot types and environments
  - a separate system selects goals for bots
  - while playing plans are created for each bot using A* and they begin executing them
    - If an action in the plan becomes unexecutable (loses precondition) then we replan

- Advantages
  - Smarter A.I. than previous reactive approaches
  - Easier to implement and maintain
  - Separation of goals and actions from implementation

- Disadvantages
  - No direct control over agent behavior (hard to debug, unpredictable behavior – designers hate that)
  - These days HTN and Behavior Trees are more in