Multi-Agent Systems

Decentralized Multi-Agent Path Finding

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel, Rolf Bergdoll, and Thorsten Engesser

Winter Term 2019/20

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.
- What if there is no central instance and communication of plans is impossible?

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.
- What if there is no central instance and communication of plans is impossible?
- In this setting, which we call *DMAPF*, we assume that everybody wants to achieve the common goal of reaching all destinations.

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.
- What if there is no central instance and communication of plans is impossible?
- In this setting, which we call *DMAPF*, we assume that everybody wants to achieve the common goal of reaching all destinations.

→ Each agent needs to plan decentrally.

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.
- What if there is no central instance and communication of plans is impossible?
- In this setting, which we call *DMAPF*, we assume that everybody wants to achieve the common goal of reaching all destinations.

→ Each agent needs to plan decentrally.

⇒ What kind of plans do we need to generate?

Going beyond MAPF

- In MAPF, planning is performed centrally, then the plan is communicated to all agents and execution is done decentrally.
- What if there is no central instance and communication of plans is impossible?
- In this setting, which we call *DMAPF*, we assume that everybody wants to achieve the common goal of reaching all destinations.

→ Each agent needs to plan decentrally.

⇒ What kind of plans do we need to generate?

⇒ How do we define the *joint execution* of such plans?

Implicitly coordinated plans (in a cooperative setting)

- An agent plans its own actions . . .

Implicitly coordinated plans (in a cooperative setting)

- An agent plans its own actions ...
- ...in a way to *empower* the other agents to reach the common goal.

Implicitly coordinated plans (in a cooperative setting)

- An agent plans its own actions . . .
- . . . in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.

Implicitly coordinated plans (in a cooperative setting)

UNI FREIBURG

- An agent plans its own actions ...
- ...in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.
- We consider one possibility for the other agent to continue the plan, i.e., the plan will be a *linear plan*.

Implicitly coordinated plans (in a cooperative setting)

- An agent plans its own actions ...
- ...in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.
- We consider one possibility for the other agent to continue the plan, i.e., the plan will be a *linear plan*.
- We assume that plans are non-redundant, i.e., that they are *cycle-free*.

Implicitly coordinated plans (in a cooperative setting)

- An agent plans its own actions ...
- ...in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.
- We consider one possibility for the other agent to continue the plan, i.e., the plan will be a *linear plan*.
- We assume that plans are non-redundant, i.e., that they are *cycle-free*.
- Executing such a plan will thus never lead to a *dead end*, i.e., a state from which the other agents cannot reach the common goal.

Implicitly coordinated plans (in a cooperative setting)

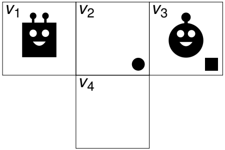
- An agent plans its own actions . . .
- . . . in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.
- We consider one possibility for the other agent to continue the plan, i.e., the plan will be a *linear plan*.
- We assume that plans are non-redundant, i.e., that they are *cycle-free*.
- Executing such a plan will thus never lead to a *dead end*, i.e., a state from which the other agents cannot reach the common goal.
- However, almost certainly, agents will come up with different (perhaps conflicting) plans.

Implicitly coordinated plans (in a cooperative setting)

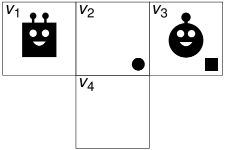
UNI FREIBURG

- An agent plans its own actions . . .
- . . . in a way to *empower* the other agents to reach the common goal.
- This implies to plan for the other agents.
- We consider one possibility for the other agent to continue the plan, i.e., the plan will be a *linear plan*.
- We assume that plans are non-redundant, i.e., that they are *cycle-free*.
- Executing such a plan will thus never lead to a *dead end*, i.e., a state from which the other agents cannot reach the common goal.
- However, almost certainly, agents will come up with different (perhaps conflicting) plans.
- How do we define joint execution of such conflicting plans?

Example: Two implicitly coordinated plans

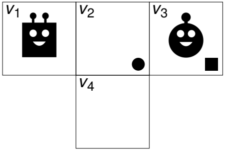


Example: Two implicitly coordinated plans



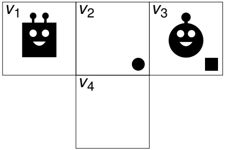
How to solve the problem?

Example: Two implicitly coordinated plans



How to solve the problem?

$$\pi_C = \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$



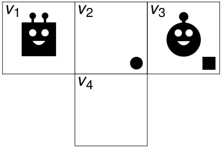
How to solve the problem?

$$
\begin{aligned}
\pi_C &= \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle \\
\pi_S &= \langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), \\
&\quad (S, v_2, v_3), (C, v_1, v_2) \rangle
\end{aligned}
$$

Joint execution

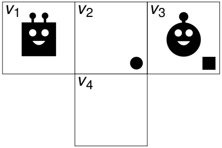
- Let us assume, all agents have planed and a subset of them came up with a *family of plans* $(\pi_i)_{i \in A}$.
- Among the agents that have a plan with their own action as the next action to execute, one is chosen.
- The action of the chosen agent is executed.
- Agents, which have anticipated the action, track that in their plans.
- All other agents have to *replan* from the new state.
- Since everybody has a successful plan, no acting agent will ever execute an action that leads to a dead end.

Example execution



Planning, executing, and replanning:

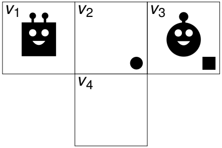
Example execution



Planning, executing, and replanning:

$$C: \quad \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$

Example execution

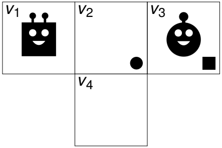


Planning, executing, and replanning:

$$C: \quad \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$

$$S: \quad \langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$$

Example execution

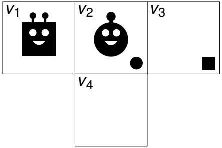


Planning, executing, and replanning:

$C: \quad \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

$S: \quad \langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

Example execution

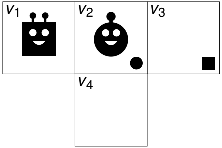


Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$

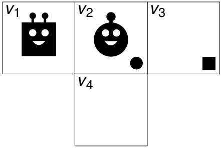
Example execution



Planning, executing, and replanning:

$$C: \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$
$$S: \langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$$

Example execution



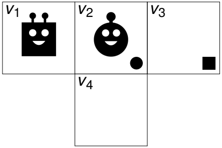
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



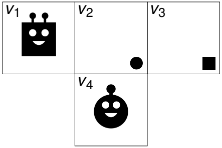
Planning, executing, and replanning:

$$C: \quad \langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$

$$S: \quad \langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$$

$$C: \quad \langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$$

Example execution



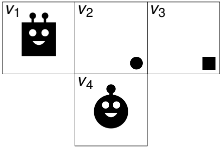
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



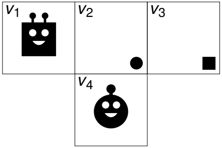
Planning, executing, and replanning:

C: $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S: $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C: $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



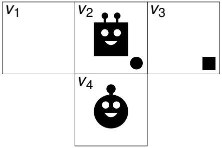
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



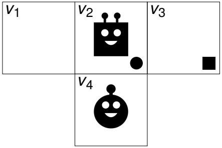
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



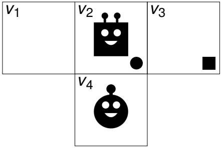
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



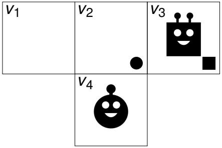
Planning, executing, and replanning:

C : $\langle(C, v_3, v_2),(C, v_2, v_4),(S, v_1, v_2),(S, v_2, v_3),(C, v_4, v_2)\rangle$

S : $\langle(S, v_1, v_2),(S, v_2, v_4),(C, v_3, v_2),(C, v_2, v_1),(S, v_4, v_2),$
$(S, v_2, v_3),(C, v_1, v_2)\rangle$

C : $\langle(C, v_2, v_4),(S, v_1, v_2),(S, v_2, v_3),(C, v_4, v_2)\rangle$

Example execution



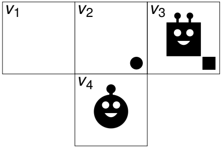
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2),$
$(S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution

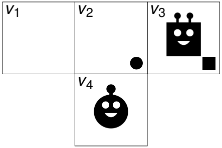


Planning, executing, and replanning:

C : $\langle(C,v_3,v_2),(C,v_2,v_4),(S,v_1,v_2),(S,v_2,v_3),(C,v_4,v_2)\rangle$

S : $\langle(S,v_1,v_2),(S,v_2,v_4),(C,v_3,v_2),(C,v_2,v_1),(S,v_4,v_2),$
$(S,v_2,v_3),(C,v_1,v_2)\rangle$

C : $\langle(C,v_2,v_4),(S,v_1,v_2),(S,v_2,v_3),(C,v_4,v_2)\rangle$



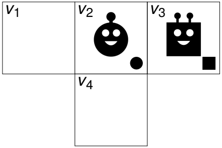
Planning, executing, and replanning:

C : $\langle(C,v_3,v_2),(C,v_2,v_4),(S,v_1,v_2),(S,v_2,v_3),(C,v_4,v_2)\rangle$

S : $\langle(S,v_1,v_2),(S,v_2,v_4),(C,v_3,v_2),(C,v_2,v_1),(S,v_4,v_2),$
$(S,v_2,v_3),(C,v_1,v_2)\rangle$

C : $\langle(C,v_2,v_4),(S,v_1,v_2),(S,v_2,v_3),(C,v_4,v_2)\rangle$

Example execution



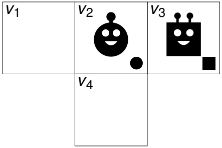
Planning, executing, and replanning:

C : $\langle (C, v_3, v_2), (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

S : $\langle (S, v_1, v_2), (S, v_2, v_4), (C, v_3, v_2), (C, v_2, v_1), (S, v_4, v_2), (S, v_2, v_3), (C, v_1, v_2) \rangle$

C : $\langle (C, v_2, v_4), (S, v_1, v_2), (S, v_2, v_3), (C, v_4, v_2) \rangle$

Example execution



Planning, executing, and replanning:

C: $\langle(C, v_3, v_2),(C, v_2, v_4),(S, v_1, v_2),(S, v_2, v_3),(C, v_4, v_2)\rangle$

S: $\langle(S, v_1, v_2),(S, v_2, v_4),(C, v_3, v_2),(C, v_2, v_1),(S, v_4, v_2),$
$(S, v_2, v_3),(C, v_1, v_2)\rangle$

C: $\langle(C, v_2, v_4),(S, v_1, v_2),(S, v_2, v_3),(C, v_4, v_2)\rangle$

Done!

Lazy and eager agents

What can go wrong?

Lazy and eager agents

What can go wrong?

- Agents could be *lazy*: Sometimes they choose a plan where they expect that another agent should act, although they could act.

→ Agents may wait forever for each other to act (dish washing dilemma).

Lazy and eager agents

What can go wrong?

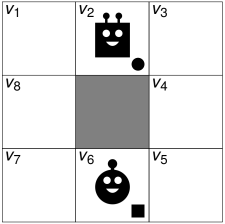
- Agents could be *lazy*: Sometimes they choose a plan where they expect that another agent should act, although they could act.

→ Agents may wait forever for each other to act (dish washing dilemma).

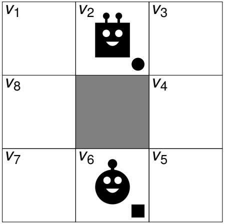
- Agents could be *eager*: If agents could act (without creating a cycle or a dead end), they choose to act.

→ Agents might create cyclic executions (without creating plans that are cyclic), leading to *infinite executions*.

Example for infinite execution

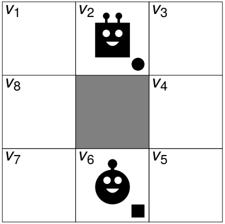


Example for infinite execution



π_1 (S initially): $\langle\ (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

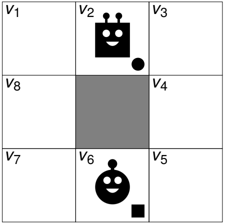
Example for infinite execution



π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

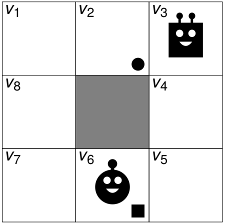
Example for infinite execution



π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

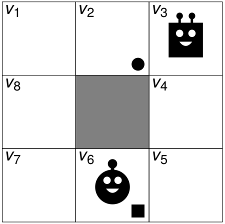
Example for infinite execution



π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

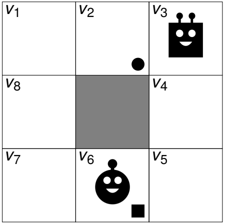
Example for infinite execution



π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

Example for infinite execution

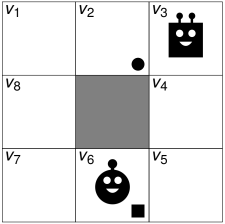


π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle\, (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

Example for infinite execution

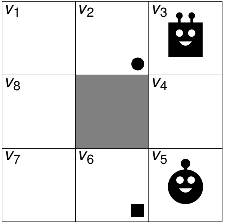


π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

Example for infinite execution

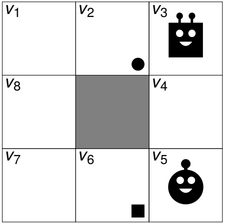


π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

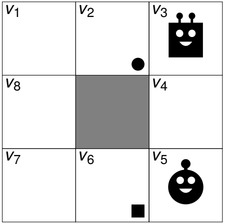
π_3 (C after (S, v_2, v_3)): $\langle (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

Example for infinite execution



π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$
π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$
π_3 (C after (S, v_2, v_3)): $\langle (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

Example for infinite execution



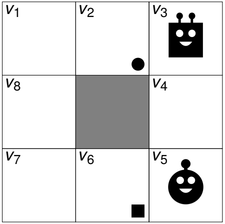
π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle\, (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

π_4 (S after (C, v_6, v_5)): $\langle\, (S, v_3, v_2), (S, v_2, v_1), (S, v_1, v_8), (S, v_8, v_7), \ldots \rangle$

Example for infinite execution



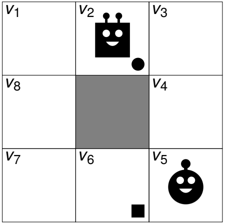
π_1 (S initially): $\langle\ (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots\rangle$

π_2 (C initially): $\langle\ (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots\rangle$

π_3 (C after (S, v_2, v_3)): $\langle\ (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots\rangle$

π_4 (S after (C, v_6, v_5)): $\langle\ (S, v_3, v_2), (S, v_2, v_1), (S, v_1, v_8), (S, v_8, v_7), \ldots\rangle$

Example for infinite execution



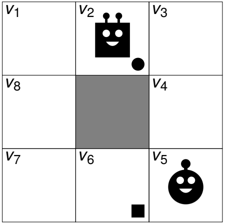
π_1 (S initially): $\langle\, (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

π_2 (C initially): $\langle\, (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle\, (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

π_4 (S after (C, v_6, v_5)): $\langle\, (S, v_3, v_2), (S, v_2, v_1), (S, v_1, v_8), (S, v_8, v_7), \ldots \rangle$

Example for infinite execution



π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

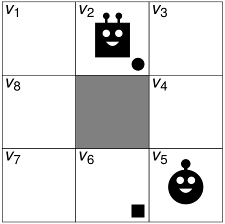
π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

π_4 (S after (C, v_6, v_5)): $\langle (S, v_3, v_2), (S, v_2, v_1), (S, v_1, v_8), (S, v_8, v_7), \ldots \rangle$

π_5 (C after (S, v_3, v_2)): $\langle (C, v_5, v_6), (C, v_6, v_7), (C, v_7, v_8), (C, v_8, v_1), \ldots \rangle$

Example for infinite execution



π_1 (S initially): $\langle\ (S,v_2,v_3),(S,v_3,v_4),(S,v_4,v_5),(C,v_6,v_7),\ldots\rangle$

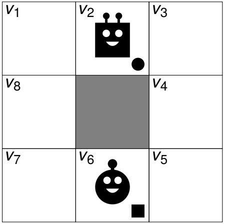
π_2 (C initially): $\langle\ (C,v_6,v_5),(C,v_5,v_4),(C,v_4,v_3),(S,v_2,v_1),\ldots\rangle$

π_3 (C after (S,v_2,v_3)): $\langle\ (C,v_6,v_5),(C,v_5,v_4),\underline{(S,v_3,v_2)},(C,v_4,v_3),\ldots\rangle$

π_4 (S after (C,v_6,v_5)): $\langle\ (S,v_3,v_2),(S,v_2,v_1),(S,v_1,v_8),(S,v_8,v_7),\ldots\rangle$

π_5 (C after (S,v_3,v_2)): $\langle\ (C,v_5,v_6),(C,v_6,v_7),(C,v_7,v_8),(C,v_8,v_1),\ldots\rangle$

Example for infinite execution



π_1 (S initially): $\langle (S, v_2, v_3), (S, v_3, v_4), (S, v_4, v_5), (C, v_6, v_7), \ldots \rangle$

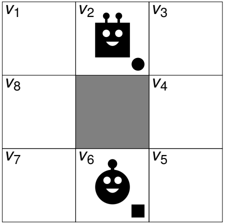
π_2 (C initially): $\langle (C, v_6, v_5), (C, v_5, v_4), (C, v_4, v_3), (S, v_2, v_1), \ldots \rangle$

π_3 (C after (S, v_2, v_3)): $\langle (C, v_6, v_5), (C, v_5, v_4), \underline{(S, v_3, v_2)}, (C, v_4, v_3), \ldots \rangle$

π_4 (S after (C, v_6, v_5)): $\langle (S, v_3, v_2), (S, v_2, v_1), (S, v_1, v_8), (S, v_8, v_7), \ldots \rangle$

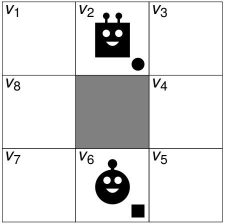
π_5 (C after (S, v_3, v_2)): $\langle (C, v_5, v_6), (C, v_6, v_7), (C, v_7, v_8), (C, v_8, v_1), \ldots \rangle$

Example for infinite execution



π_1 (S initially): $\langle\, (S,v_2,v_3),(S,v_3,v_4),(S,v_4,v_5),(C,v_6,v_7),\ldots\rangle$
π_2 (C initially): $\langle\, (C,v_6,v_5),(C,v_5,v_4),(C,v_4,v_3),(S,v_2,v_1),\ldots\rangle$
π_3 (C after (S,v_2,v_3)): $\langle\, (C,v_6,v_5),(C,v_5,v_4),\underline{(S,v_3,v_2)},(C,v_4,v_3),\ldots\rangle$
π_4 (S after (C,v_6,v_5)): $\langle\, (S,v_3,v_2),(S,v_2,v_1),(S,v_1,v_8),(S,v_8,v_7),\ldots\rangle$
π_5 (C after (S,v_3,v_2)): $\langle\, (C,v_5,v_6),(C,v_6,v_7),(C,v_7,v_8),(C,v_8,v_1),\ldots\rangle$

Example for infinite execution



π_1 (S initially): $\langle (S,v_2,v_3),(S,v_3,v_4),(S,v_4,v_5),(C,v_6,v_7),\ldots\rangle$

π_2 (C initially): $\langle (C,v_6,v_5),(C,v_5,v_4),(C,v_4,v_3),(S,v_2,v_1),\ldots\rangle$

π_3 (C after (S,v_2,v_3)): $\langle (C,v_6,v_5),(C,v_5,v_4),\underline{(S,v_3,v_2)},(C,v_4,v_3),\ldots\rangle$

π_4 (S after (C,v_6,v_5)): $\langle (S,v_3,v_2),(S,v_2,v_1),\underline{(S,v_1,v_8)},(S,v_8,v_7),\ldots\rangle$

π_5 (C after (S,v_3,v_2)): $\langle (C,v_5,v_6),(C,v_6,v_7),(C,v_7,v_8),(C,v_8,v_1),\ldots\rangle$

π_5 (S after (C,v_5,v_6)): $\langle (S,v_2,v_3),\ldots\rangle$

Optimally eager agents

- Eager agents avoid *deadlocks*, however they are *hyper-active*.
- They might even move away from their destination!
- So, let force them to be smart: They should generate only optimal plans . . . and among those optimal plans they should also be eager.
- In our previous example: After the square agent moved right, the circle agent will choose to move left!

→ Does it always work out?

Optimally eager agents are always successful

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Optimally eager agents are always successful

UNI FREIBURG

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

□

Optimally eager agents are always successful

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

k=0: Obviously true.

□

Optimally eager agents are always successful

UNI FREIBURG

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

k=0: Obviously true.

Assume the claim is true for k. Consider a DMAPF instance such that there exists a shortest plan of length $k+1$.

□

Optimally eager agents are always successful

UNI FREIBURG

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

k=0: Obviously true.

Assume the claim is true for k. Consider a DMAPF instance such that there exists a shortest plan of length $k+1$. Because the agents are eager, at least one agent wants to move.

□

Optimally eager agents are always successful

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

k=0: Obviously true.

Assume the claim is true for k. Consider a DMAPF instance such that there exists a shortest plan of length $k+1$. Because the agents are eager, at least one agent wants to move. One agent will move (according to an optimal plan) and by this reduce the necessary number of steps by one.

□

Optimally eager agents are always successful

Theorem

Optimally eager agents are always successful on all solvable DMAPF instances.

Proof.

By induction over the length of a shortest plan k.

k=0: Obviously true.

Assume the claim is true for k. Consider a DMAPF instance such that there exists a shortest plan of length $k+1$. Because the agents are eager, at least one agent wants to move. One agent will move (according to an optimal plan) and by this reduce the necessary number of steps by one. Hence, we have now an instance with plan length k and the induction hypothesis applies. □

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.
- Is it possible to solve the problem using only polynomial time?

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.
- Is it possible to solve the problem using only polynomial time?
- *Conservative replanning*: Always start at the initial state and consider the already executed movements as a prefix of the new plan.

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.
- Is it possible to solve the problem using only polynomial time?
- *Conservative replanning*: Always start at the initial state and consider the already executed movements as a prefix of the new plan.

→ Avoids infinite executions because plans have to be cycle-free.

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.
- Is it possible to solve the problem using only polynomial time?
- *Conservative replanning*: Always start at the initial state and consider the already executed movements as a prefix of the new plan.

→ Avoids infinite executions because plans have to be cycle-free.

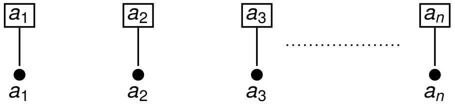
⇒ The agents might visit the entire state space before terminating.

Conservative replanning

- Optimally eager agents have to solve a sequence of NP-hard problems.
- Is it possible to solve the problem using only polynomial time?
- *Conservative replanning*: Always start at the initial state and consider the already executed movements as a prefix of the new plan.

→ Avoids infinite executions because plans have to be cycle-free.

⇒ The agents might visit the entire state space before terminating.



Other ways to coordinate?

- One way to avoid NP-hardness or exponentially longer plans is to use *approximation algorithms*, but we know of none.

Other ways to coordinate?

- One way to avoid NP-hardness or exponentially longer plans is to use *approximation algorithms*, but we know of none.
- Is it possible to use the rule-based algorithms (which are polynomial)?

Other ways to coordinate?

- One way to avoid NP-hardness or exponentially longer plans is to use *approximation algorithms*, but we know of none.
- Is it possible to use the rule-based algorithms (which are polynomial)?
- Assume that everybody uses the same algorithm: Of course, the agents would act in coordinated way, but this more like central planning.

Other ways to coordinate?

- One way to avoid NP-hardness or exponentially longer plans is to use *approximation algorithms*, but we know of none.
- Is it possible to use the rule-based algorithms (which are polynomial)?
- Assume that everybody uses the same algorithm: Of course, the agents would act in coordinated way, but this more like central planning.
- If the agents may use different algorithms, then it is not clear how to avoid cyclic executions.

Other ways to coordinate?

- One way to avoid NP-hardness or exponentially longer plans is to use *approximation algorithms*, but we know of none.
- Is it possible to use the rule-based algorithms (which are polynomial)?
- Assume that everybody uses the same algorithm: Of course, the agents would act in coordinated way, but this more like central planning.
- If the agents may use different algorithms, then it is not clear how to avoid cyclic executions.
- Conservative replanning is not helpful in this context, because the executed actions might not be a prefix of a valid plan!

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.
- All agents know their own destinations, but these are *not common knowledge* any longer.

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.
- All agents know their own destinations, but these are *not common knowledge* any longer.
- For each agent, there exists a *set of possible destinations*, which are *common knowledge*.

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.
- All agents know their own destinations, but these are *not common knowledge* any longer.
- For each agent, there exists a *set of possible destinations*, which are *common knowledge*.
- All agents plan and re-plan without communicating with their peers.

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.
- All agents know their own destinations, but these are *not common knowledge* any longer.
- For each agent, there exists a *set of possible destinations*, which are *common knowledge*.
- All agents plan and re-plan without communicating with their peers.
- A *success announcement action* becomes necessary, which the agents may use to announce that they have reached their destination (and after that they are not allowed to move anymore).

MAPF/DU: MAPF under destination uncertainty

MAPF under *destination uncertainty* (MAPF/DU):

- The *common goal* of all agents is that everybody reaches its destination.
- All agents know their own destinations, but these are *not common knowledge* any longer.
- For each agent, there exists a *set of possible destinations*, which are *common knowledge*.
- All agents plan and re-plan without communicating with their peers.
- A *success announcement action* becomes necessary, which the agents may use to announce that they have reached their destination (and after that they are not allowed to move anymore).

→ Models multi-robot interactions without communication

MAPF/DU: Conceptual problems

UNI FREIBURG

- We need a *solution concept* for the agents: *implicitly coordinated branching plans*.

MAPF/DU: Conceptual problems

- We need a *solution concept* for the agents: *implicitly coordinated branching plans*.
- We need to find conditions that guarantee success of joint execution.

MAPF/DU: Conceptual problems

- We need a *solution concept* for the agents: *implicitly coordinated branching plans*.
- We need to find conditions that guarantee success of joint execution.
- We have to determine the computational complexity for finding plans and deciding solvability.

MAPF/DU: Conceptual problems

- We need a *solution concept* for the agents: *implicitly coordinated branching plans*.
- We need to find conditions that guarantee success of joint execution.
- We have to determine the computational complexity for finding plans and deciding solvability.

→ Since MAPF/DU is a special case of epistemic planning (initial state uncertainty which is monotonically decreasing), we can use concepts and results from this area.

MAPF/DU representation & state space

- In addition to the sets of agents A, the graph $G = (V, E)$, and the assignment of agents to nodes α, we need a function to represent the *possible destinations* $\beta : A \rightarrow 2^V$.

MAPF/DU representation & state space

- In addition to the sets of agents A, the graph $G = (V, E)$, and the assignment of agents to nodes α, we need a function to represent the *possible destinations* $\beta : A \rightarrow 2^V$.
- We assume that the set of possible destinations are pairwise disjoint (this can be relaxed, though).

MAPF/DU representation & state space

- In addition to the sets of agents A, the graph $G = (V, E)$, and the assignment of agents to nodes α, we need a function to represent the *possible destinations* $\beta : A \to 2^V$.
- We assume that the set of possible destinations are pairwise disjoint (this can be relaxed, though).
- An *objective state* is given by the pair $s = \langle \alpha, \beta \rangle$ representing the common knowledge of all agents.

MAPF/DU representation & state space

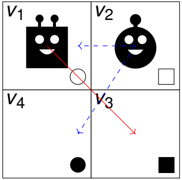
UNI FREIBURG

- In addition to the sets of agents A, the graph $G = (V, E)$, and the assignment of agents to nodes α, we need a function to represent the *possible destinations* $\beta : A \to 2^V$.
- We assume that the set of possible destinations are pairwise disjoint (this can be relaxed, though).
- An *objective state* is given by the pair $s = \langle \alpha, \beta \rangle$ representing the common knowledge of all agents.
- A *subjective state* of agent i is given by $s^i \langle \alpha, \beta, i, v \rangle$ with $v \in \beta(i)$, representing the private knowledge of agent i.

MAPF/DU representation & state space

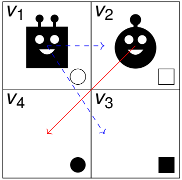
- In addition to the sets of agents A, the graph $G = (V, E)$, and the assignment of agents to nodes α, we need a function to represent the *possible destinations* $\beta : A \rightarrow 2^V$.
- We assume that the set of possible destinations are pairwise disjoint (this can be relaxed, though).
- An *objective state* is given by the pair $s = \langle \alpha, \beta \rangle$ representing the common knowledge of all agents.
- A *subjective state* of agent i is given by $s^i \langle \alpha, \beta, i, v \rangle$ with $v \in \beta(i)$, representing the private knowledge of agent i.
- A *MAPF/DU instance* is given by $\langle A, G, s_0, \alpha_* \rangle$, where $s_0 = \langle \alpha_0, \beta_0 \rangle$.

MAPF/DU: Implicitly coordinated branching plans



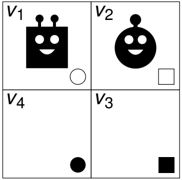
- Square agent S wants to go to v_3 and knows that circle agent C wants to go to v_1 or v_4.

MAPF/DU: Implicitly coordinated branching plans



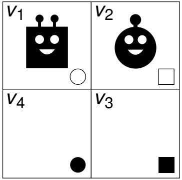
- Square agent S wants to go to v_3 and knows that circle agent C wants to go to v_1 or v_4.
- C wants to go to v_4 and knows that S wants to go to v_2 or v_3.

MAPF/DU: Implicitly coordinated branching plans



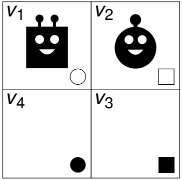
- Square agent S wants to go to v_3 and knows that circle agent C wants to go to v_1 or v_4.
- C wants to go to v_4 and knows that S wants to go to v_2 or v_3.
- Let us assume S forms a plan in which it moves in order to empower C to reach their common goal.

MAPF/DU: Implicitly coordinated branching plans



- Square agent S wants to go to v_3 and knows that circle agent C wants to go to v_1 or v_4.
- C wants to go to v_4 and knows that S wants to go to v_2 or v_3.
- Let us assume S forms a plan in which it moves in order to empower C to reach their common goal.
- S needs *shifting its perspective* in order to plan for all possible destinations of C (*branching on destinations*).

MAPF/DU: Implicitly coordinated branching plans



- Square agent S wants to go to v_3 and knows that circle agent C wants to go to v_1 or v_4.
- C wants to go to v_4 and knows that S wants to go to v_2 or v_3.
- Let us assume S forms a plan in which it moves in order to empower C to reach their common goal.
- S needs *shifting its perspective* in order to plan for all possible destinations of C (*branching on destinations*).
- Planning for C, S must *forget* about its own destination.

Branching plans: Building blocks

Branching plans consist of:

- *Movement actions*: $(\langle agent \rangle, \langle sourcenode \rangle, \langle targetnode \rangle)$, i.e., a movement of an agent
- *Success announcement*: $(\langle agent \rangle, \mathscr{S})$, after that all agents know that the agent has reached its destination and it cannot move anymore
- *Perspective shift*: $[\langle agent \rangle : \ldots]$, i.e., from here on we assume to plan with the knowledge of agent $\langle agent \rangle$. This can be unconditional or conditional on $\langle agent \rangle$'s destinations.
- *Branch on all destinations*: $(?\langle dest_1 \rangle \{\ldots\}, \ldots, ?\langle dest_n \rangle \{\ldots\})$, where all destinations of the current agent have to be listed. For each case we try to find a successful plan to reach the goal state.

Semantics of branching plans

- Movement actions modify α in the obvious way.

Semantics of branching plans

- Movement actions modify α in the obvious way.
- A success announcement of agent i (only possible if agent $\beta(i) = v$ and i is at v) transforms β to β' such that $\beta'(i) = \emptyset$ in order to signal that i cannot move anymore

Semantics of branching plans

- Movement actions modify α in the obvious way.
- A success announcement of agent i (only possible if agent $\beta(i) = v$ and i is at v) transforms β to β' such that $\beta'(i) = \emptyset$ in order to signal that i cannot move anymore
- A perspective shift from i to j with subsequent branching on destinations transforms the subjective state $s^i = \langle \alpha, \beta, i, v_i \rangle$ to a set of subjective states $s^{j_k} = \langle \alpha, \beta, j, v_{j_k} \rangle$ with all $v_{j_k} \in \beta(j)$.

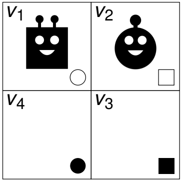
Semantics of branching plans

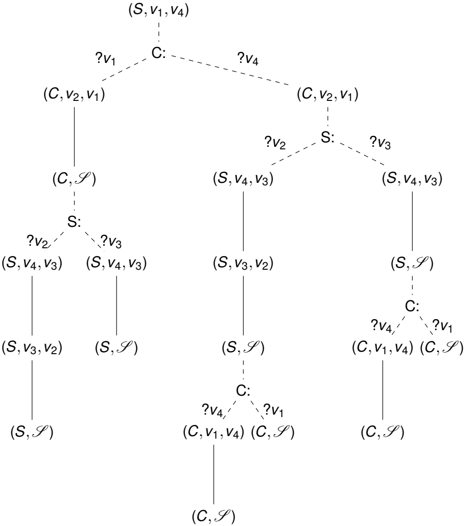
- Movement actions modify α in the obvious way.
- A success announcement of agent i (only possible if agent $\beta(i) = v$ and i is at v) transforms β to β' such that $\beta'(i) = \emptyset$ in order to signal that i cannot move anymore
- A perspective shift from i to j with subsequent branching on destinations transforms the subjective state $s^i = \langle \alpha, \beta, i, v_i \rangle$ to a set of subjective states $s^{j_k} = \langle \alpha, \beta, j, v_{j_k} \rangle$ with all $v_{j_k} \in \beta(j)$.
- A perspective shift from i to j without subsequent branching on destinations induces the same transformation, but enforces that the subsequent plans are the same for all states subjective states s^{j_k}.

Semantics of branching plans

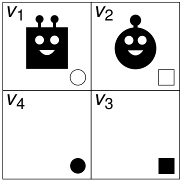
- Movement actions modify α in the obvious way.
- A success announcement of agent i (only possible if agent $\beta(i) = v$ and i is at v) transforms β to β' such that $\beta'(i) = \emptyset$ in order to signal that i cannot move anymore
- A perspective shift from i to j with subsequent branching on destinations transforms the subjective state $s^i = \langle \alpha, \beta, i, v_i \rangle$ to a set of subjective states $s^{j_k} = \langle \alpha, \beta, j, v_{j_k} \rangle$ with all $v_{j_k} \in \beta(j)$.
- A perspective shift from i to j without subsequent branching on destinations induces the same transformation, but enforces that the subsequent plans are the same for all states subjective states s^{j_k}.
- Note: After a perspective shift to j, only j can move and announce success!

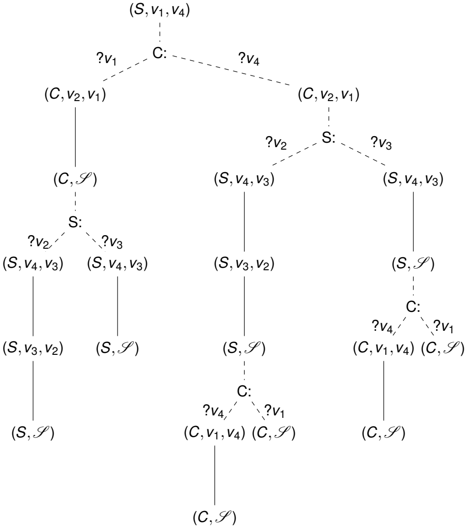
Branching plan: Example



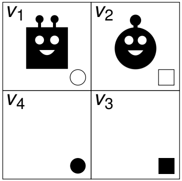


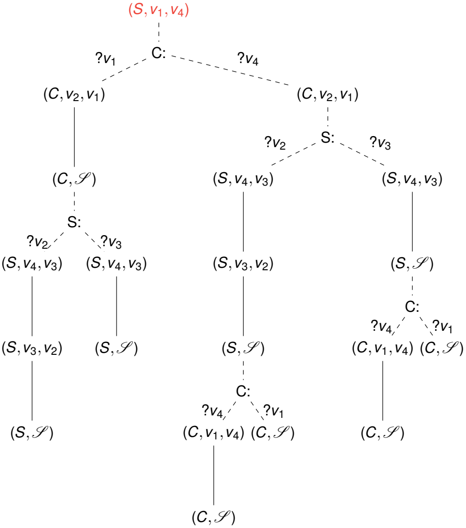
Branching plan: Subjective execution example



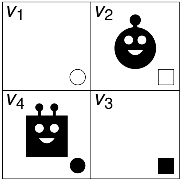


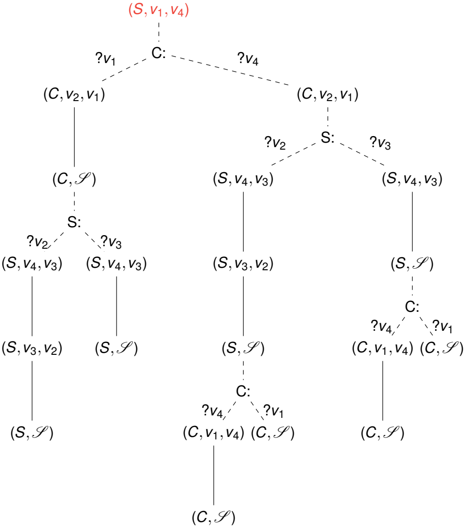
Branching plan: Subjective execution example



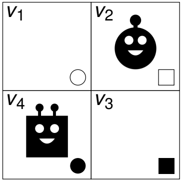


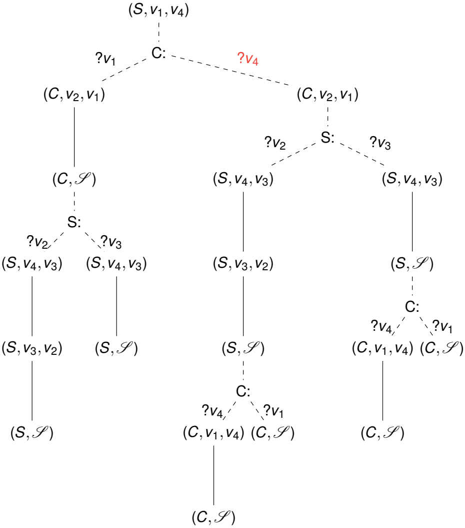
Branching plan: Subjective execution example



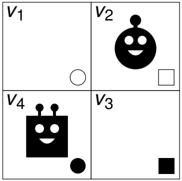


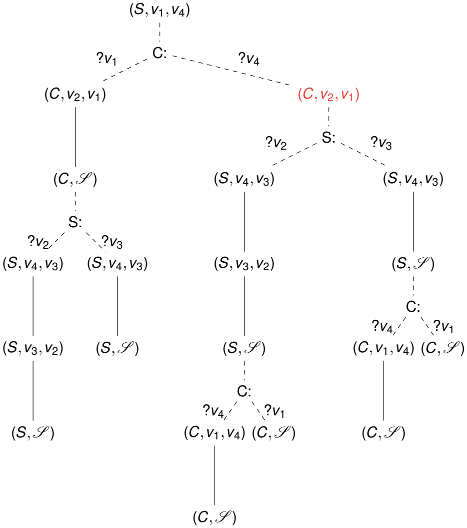
Branching plan: Subjective execution example



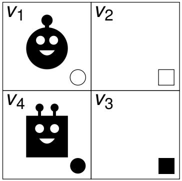


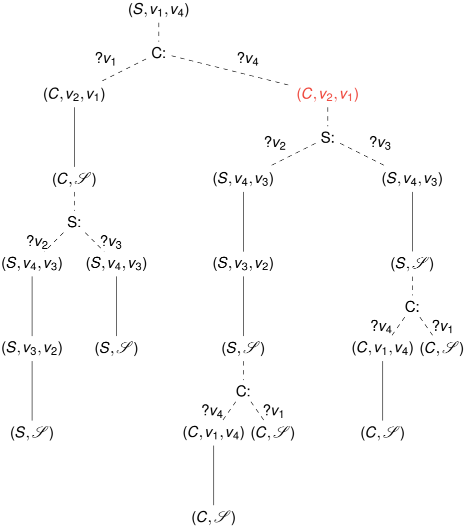
Branching plan: Subjective execution example



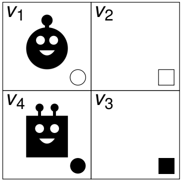


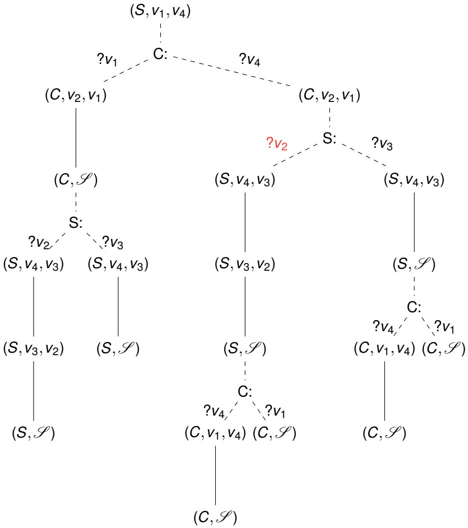
Branching plan: Subjective execution example



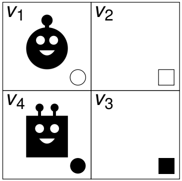


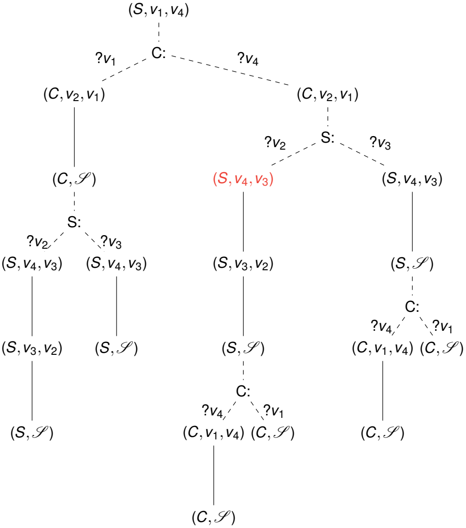
Branching plan: Subjective execution example



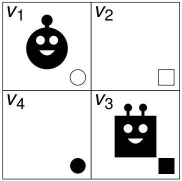


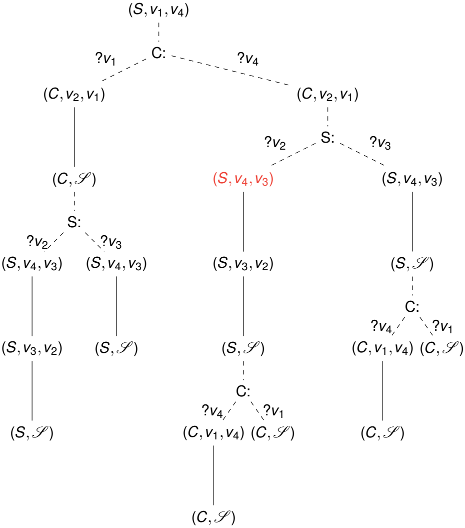
Branching plan: Subjective execution example



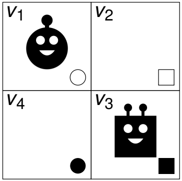


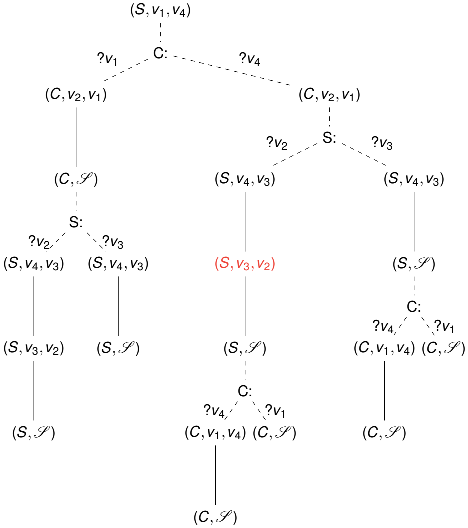
Branching plan: Subjective execution example



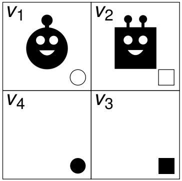


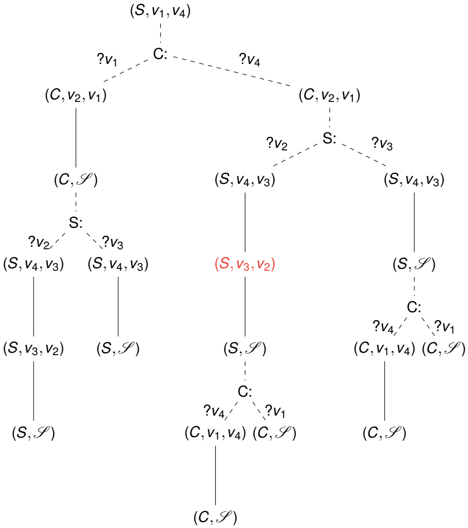
Branching plan: Subjective execution example



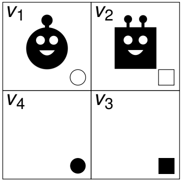


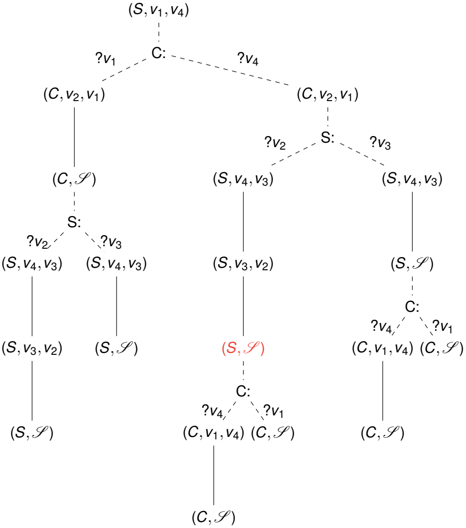
Branching plan: Subjective execution example



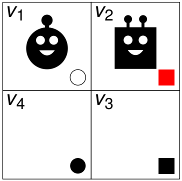


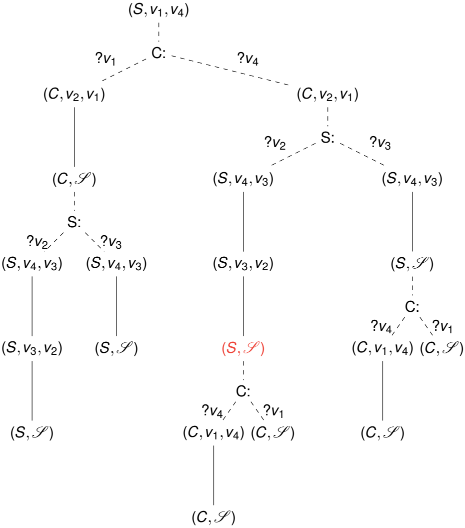
Branching plan: Subjective execution example



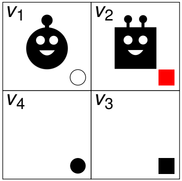


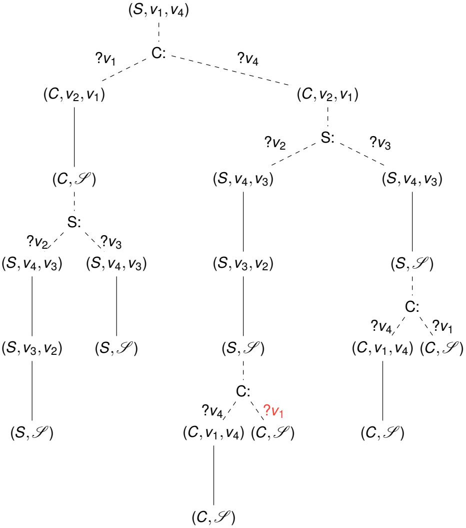
Branching plan: Subjective execution example



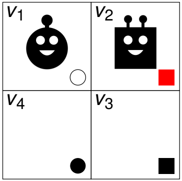


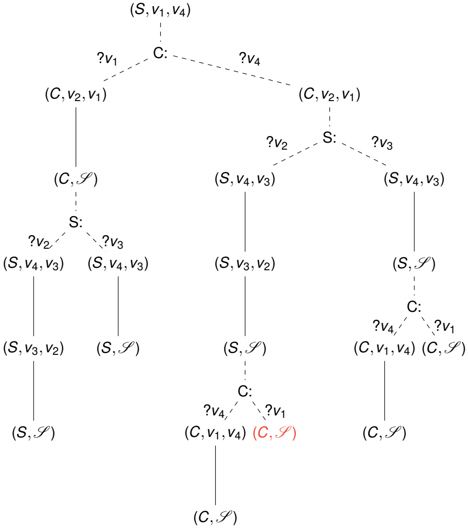
Branching plan: Subjective execution example



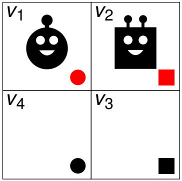


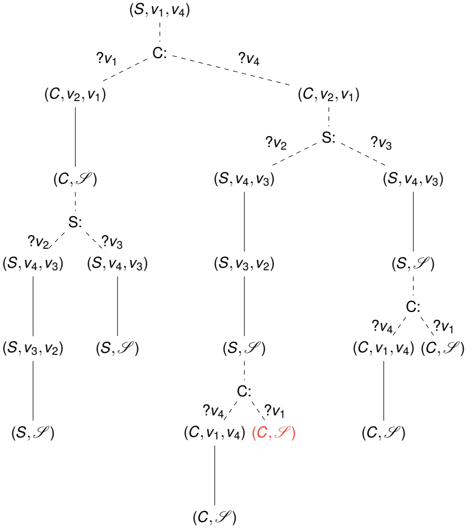
Branching plan: Subjective execution example





Branching plan: Subjective execution example





Strong plans

Similar to the notion of strong plans in non-deterministic single-agent planning, we define *i-strong plans* for an agent *i* to be:

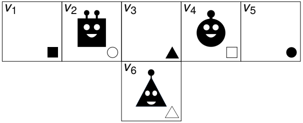
- *cycle-free*, i.e., not visiting the same objective state twice;
- *always successful*, i.e. always ending up in a state such that all agents have announced success;
- *covering*, i.e., for all combinations of possible destinations of agents different from *i*, success can be reached.

Subjectively and objectively strong plans

- A plan is called *subjectively strong* if it is i-strong for some agent i.
- A plan is called *objectively strong* if it is i-strong for each agent i.
- An instance is *objectively* or *subjectively solvable* if there exists an objectively or subjectively strong plan, respectively.

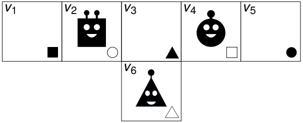
Subjectively and objectively strong plans

- A plan is called *subjectively strong* if it is i-strong for some agent i.
- A plan is called *objectively strong* if it is i-strong for each agent i.
- An instance is *objectively* or *subjectively solvable* if there exists an objectively or subjectively strong plan, respectively.



Subjectively and objectively strong plans

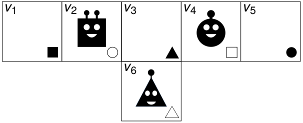
- A plan is called *subjectively strong* if it is i-strong for some agent i.
- A plan is called *objectively strong* if it is i-strong for each agent i.
- An instance is *objectively* or *subjectively solvable* if there exists an objectively or subjectively strong plan, respectively.



→ There does not exist a T-strong plan, but an S- and a C-strong plan.

Subjectively and objectively strong plans

- A plan is called *subjectively strong* if it is i-strong for some agent i.
- A plan is called *objectively strong* if it is i-strong for each agent i.
- An instance is *objectively* or *subjectively solvable* if there exists an objectively or subjectively strong plan, respectively.

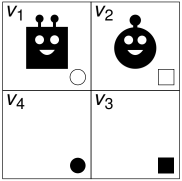


→ There does not exist a T-strong plan, but an S- and a C-strong plan.

- Difference between subjective and objective solvability concerns only the first acting agent!

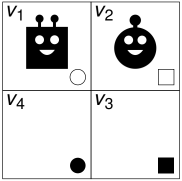
Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.



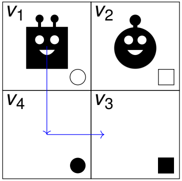
Structure of strong plans: Stepping stones

- A *stepping stone* for agent i is a state in which i can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an i-strong plan to solve the resulting states.
- S can create a stepping stone for C by moving from v_1 via v_4 to v_3.



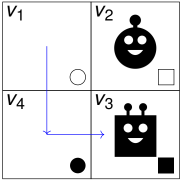
Structure of strong plans: Stepping stones

- A *stepping stone* for agent i is a state in which i can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an i-strong plan to solve the resulting states.
- S can create a stepping stone for C by moving from v_1 via v_4 to v_3.



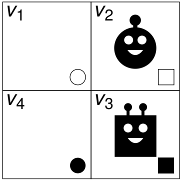
Structure of strong plans: Stepping stones

- A *stepping stone* for agent i is a state in which i can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an i-strong plan to solve the resulting states.
- S can create a stepping stone for C by moving from v_1 via v_4 to v_3.



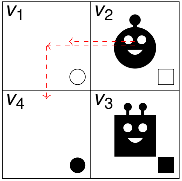
Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.
- *S* can create a stepping stone for *C* by moving from v_1 via v_4 to v_3.
- *C* can now move to v_1 or v_4 and announce success.



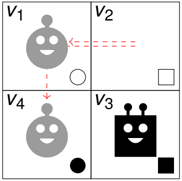
Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.
- *S* can create a stepping stone for *C* by moving from v_1 via v_4 to v_3.
- *C* can now move to v_1 or v_4 and announce success.



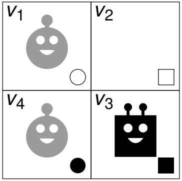
Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.
- *S* can create a stepping stone for *C* by moving from v_1 via v_4 to v_3.
- *C* can now move to v_1 or v_4 and announce success.



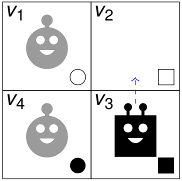
Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.
- *S* can create a stepping stone for *C* by moving from v_1 via v_4 to v_3.
- *C* can now move to v_1 or v_4 and announce success.
- In each case, *S* can move afterwards to its destination (or stay) and announce success.



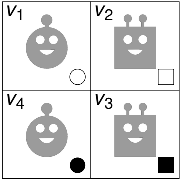
Structure of strong plans: Stepping stones

- A *stepping stone* for agent i is a state in which i can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an i-strong plan to solve the resulting states.
- S can create a stepping stone for C by moving from v_1 via v_4 to v_3.
- C can now move to v_1 or v_4 and announce success.
- In each case, S can move afterwards to its destination (or stay) and announce success.



Structure of strong plans: Stepping stones

- A *stepping stone* for agent *i* is a state in which *i* can move to each of its possible destinations, announcing success, and afterwards, for each possible destination, there exists an *i*-strong plan to solve the resulting states.
- *S* can create a stepping stone for *C* by moving from v_1 via v_4 to v_3.
- *C* can now move to v_1 or v_4 and announce success.
- In each case, *S* can move afterwards to its destination (or stay) and announce success.



Stepping Stone Theorem

Theorem

Given an i-solvable MAPF/DU instance, there exists an i-strong branching plan such that the only branching points are those utilizing stepping stones.

Stepping Stone Theorem

Theorem

Given an i-solvable MAPF/DU instance, there exists an i-strong branching plan such that the only branching points are those utilizing stepping stones.

Proof sketch.

Remove non-stepping stone branching points by picking one branch without success announcement. □

UNI FREIBURG

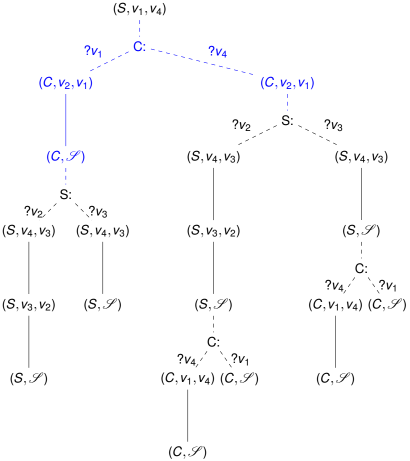
Theorem

Given an i-solvable MAPF/DU instance, there exists an i-strong branching plan such that the only branching points are those utilizing stepping stones.

Proof sketch.

Remove non-stepping stone branching points by picking one branch without success announcement. □

Proof by example



Stepping Stone Theorem

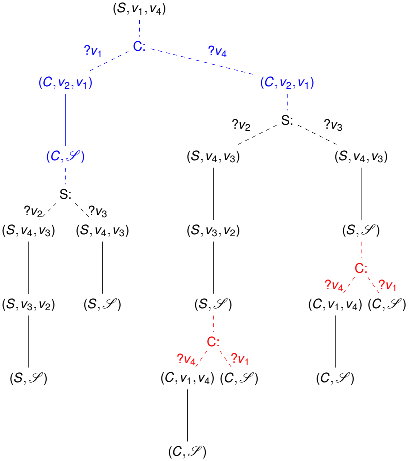
Theorem

Given an i-solvable MAPF/DU instance, there exists an i-strong branching plan such that the only branching points are those utilizing stepping stones.

Proof sketch.

Remove non-stepping stone branching points by picking one branch without success announcement. □

Proof by example



Stepping Stone Theorem

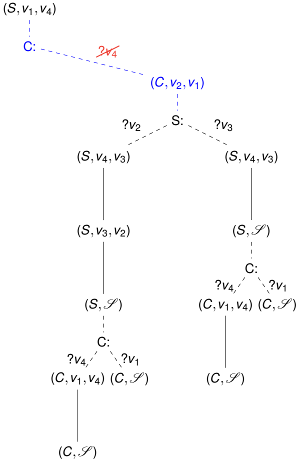
Theorem

Given an i-solvable MAPF/DU instance, there exists an i-strong branching plan such that the only branching points are those utilizing stepping stones.

Proof sketch.

Remove non-stepping stone branching points by picking one branch without success announcement. □

Proof by example



Execution cost

The *execution cost* of a branching plan is the number of atomic actions of the longest execution trace.

Execution cost

The *execution cost* of a branching plan is the number of atomic actions of the longest execution trace.

Theorem

Given an i-solvable MAPF/DU instance over a graph $G = (V, E)$, then there exists an i-strong branching plan with execution cost bounded by $O(|V|^4)$.

Execution cost

The *execution cost* of a branching plan is the number of atomic actions of the longest execution trace.

Theorem

Given an i-solvable MAPF/DU instance over a graph $G = (V, E)$, then there exists an i-strong branching plan with execution cost bounded by $O(|V|^4)$.

Proof sketch.

Direct consequence of the stepping stone theorem and the maximal number of movements in the MAPF problem. □

Joint execution and execution guarantees

- Joint execution is defined similarly to the fully observable case: One agent is chosen; afterwards the plan is tracked or the agent has to replan.

Joint execution and execution guarantees

- Joint execution is defined similarly to the fully observable case: One agent is chosen; afterwards the plan is tracked or the agent has to replan.
- In the MAPF/DU framework not all agents might have a plan initially!

Joint execution and execution guarantees

- Joint execution is defined similarly to the fully observable case: One agent is chosen; afterwards the plan is tracked or the agent has to replan.
- In the MAPF/DU framework not all agents might have a plan initially!
- One might hope that optimally eager agents are always successful.

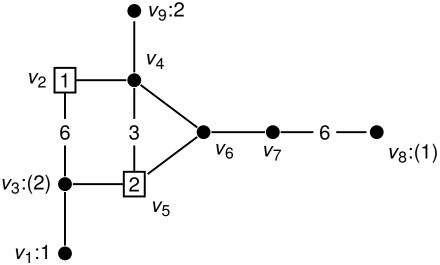
Joint execution and execution guarantees

- Joint execution is defined similarly to the fully observable case: One agent is chosen; afterwards the plan is tracked or the agent has to replan.
- In the MAPF/DU framework not all agents might have a plan initially!
- One might hope that optimally eager agents are always successful.
- In epistemic planning this was proven to be true only in the *uniform knowledge* case.

Joint execution and execution guarantees

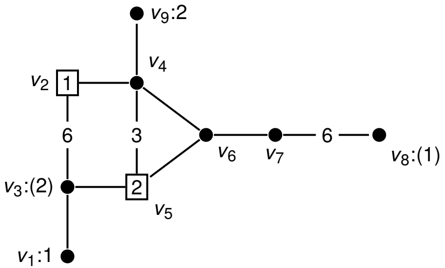
- Joint execution is defined similarly to the fully observable case: One agent is chosen; afterwards the plan is tracked or the agent has to replan.
- In the MAPF/DU framework not all agents might have a plan initially!
- One might hope that optimally eager agents are always successful.
- In epistemic planning this was proven to be true only in the *uniform knowledge* case.
- We do not have uniform knowledge … and indeed, execution cycles cannot be excluded.

A counter example



A number on an edge means that there are as many nodes on a line.

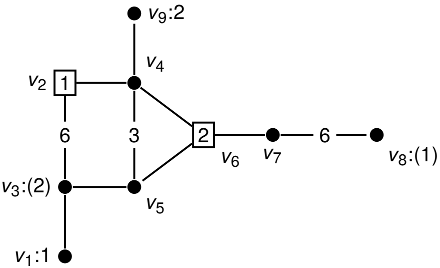
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.

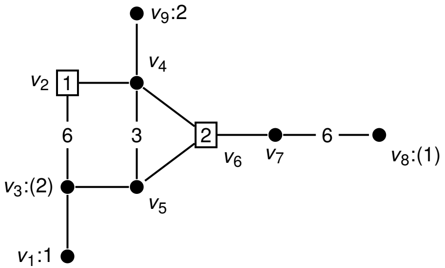
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.

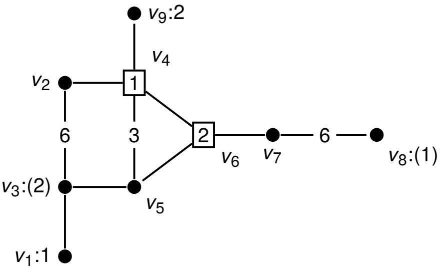
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.

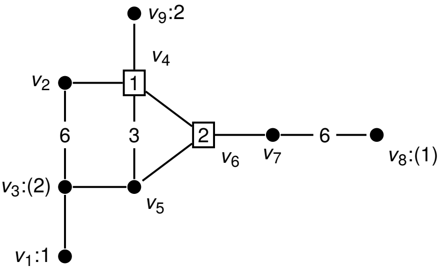
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.

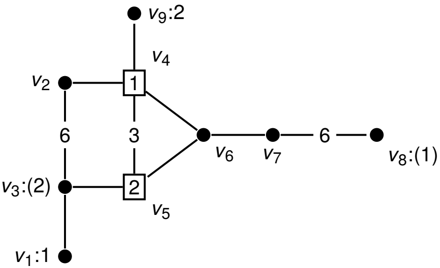
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.
- Agent 2 has then a shortest eager plan moving first to v_5.

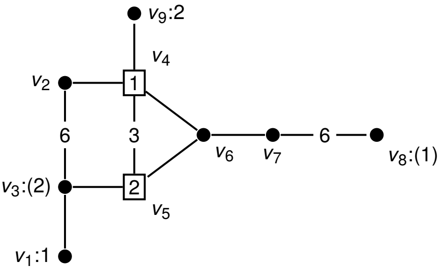
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.
- Agent 2 has then a shortest eager plan moving first to v_5.

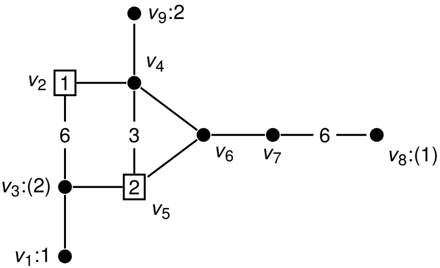
A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.
- Agent 2 has then a shortest eager plan moving first to v_5.
- Agent 1 has then a shortest eager plan moving first to v_2.

A counter example



A number on an edge means that there are as many nodes on a line.

- Agent 2 has a shortest eager plan moving first to v_6.
- Agent 1 has then a shortest eager plan moving first to v_4.
- Agent 2 has then a shortest eager plan moving first to v_5.
- Agent 1 has then a shortest eager plan moving first to v_2.

Conservatism

- Perhaps conservatism can help!
- Similarly to DMAPF, conservative replanning means that the already executed actions are used as a prefix in the plan to be generated.
- Differently from DMAPF, we assume that after a success announcement, the initial state is modified so that the *real destination* of the agent is known in the initial state.
- Otherwise we could not solve instances that are only subjectively solvable.

Conservative, optimally eager agents

- Conservative, eager agents are always successful, but might visit the entire state space before terminating.

Conservative, optimally eager agents

- Conservative, eager agents are always successful, but might visit the entire state space before terminating.
- Adding optimal eagerness can help to reduce the execution length.

Conservative, optimally eager agents

- Conservative, eager agents are always successful, but might visit the entire state space before terminating.
- Adding optimal eagerness can help to reduce the execution length.

Theorem

For solvable MAPF/DU instances, joint execution and replanning by conservative, optimally eager agents is always successful and the execution length is polynomial.

Conservative, optimally eager agents

- Conservative, eager agents are always successful, but might visit the entire state space before terminating.
- Adding optimal eagerness can help to reduce the execution length.

Theorem

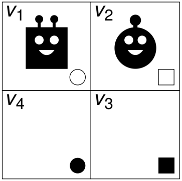
For solvable MAPF/DU instances, joint execution and replanning by conservative, optimally eager agents is always successful and the execution length is polynomial.

Proof idea.

After the second agent starts to act, all agents have an identical perspective and for this reason produce objectively strong plans with the same execution costs, which can be shown to be bounded polynomially using the stepping stone theorem. □

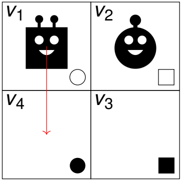
Conservative replanning example

- Assume S moves first to v_4.



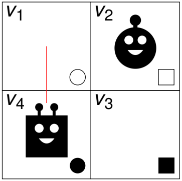
Conservative replanning example

- Assume S moves first to v_4.

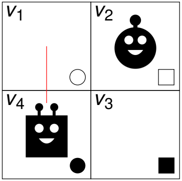


Conservative replanning example

- Assume S moves first to v_4.

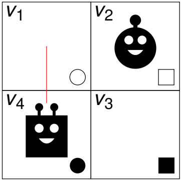


Conservative replanning example



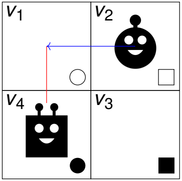
- Assume S moves first to v_4.
- Assume C re-plans. From now on, in replanning from the beginning, it has to do a perspective shift to S, because it now has to extend the partial plan starting with (S, v_4, v_1), i.e., it has to create an objectively strong plan.

Conservative replanning example



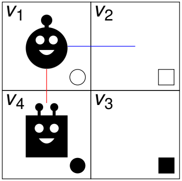
- Assume S moves first to v_4.
- Assume C re-plans. From now on, in replanning from the beginning, it has to do a perspective shift to S, because it now has to extend the partial plan starting with (S, v_4, v_1), i.e., it has to create an objectively strong plan.
- Assume that C moves now to v_1.

Conservative replanning example



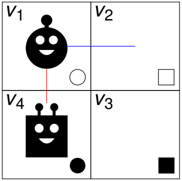
- Assume S moves first to v_4.
- Assume C re-plans. From now on, in replanning from the beginning, it has to do a perspective shift to S, because it now has to extend the partial plan starting with (S, v_4, v_1), i.e., it has to create an objectively strong plan.
- Assume that C moves now to v_1.

Conservative replanning example



- Assume S moves first to v_4.
- Assume C re-plans. From now on, in replanning from the beginning, it has to do a perspective shift to S, because it now has to extend the partial plan starting with (S, v_4, v_1), i.e., it has to create an objectively strong plan.
- Assume that C moves now to v_1.

Conservative replanning example



- Assume S moves first to v_4.
- Assume C re-plans. From now on, in replanning from the beginning, it has to do a perspective shift to S, because it now has to extend the partial plan starting with (S, v_4, v_1), i.e., it has to create an objectively strong plan.
- Assume that C moves now to v_1.
- From now on, also S has to make a perspective shift to C, effectively “forgetting” its own destination, i.e., it also has to create a objectively strong plan.

Computational Complexity: Algorithms and Turing machines

- We use Turing machines as formal models of algorithms
- This is justified, because:
  - we assume that Turing machines can compute all computable functions
  - the resource requirements (in term of time and memory) of a Turing machine are only polynomially worse than other models
- The regular type of Turing machine is the deterministic one: DTM (or simply TM)
- Often, however, we use the notion of nondeterministic TMs: NDTM

Computational Complexity: Complexity classes P and NP

Problems are categorized into complexity classes according to the requirements of computational resources:

- The class of problems decidable on deterministic Turing machines in polynomial time: P
  - Problems in P are assumed to be efficiently solvable (although this might not be true if the exponent is very large)
  - In practice, a reasonable definition
- The class of problems decidable on non-deterministic Turing machines in polynomial time, i.e., having a poly. length accepting computation for all positive instances: NP
- More classes are definable using other resource bounds on time and memory

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:
  - provide an algorithm

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:
  - provide an algorithm
  - show that the resource bounds are respected

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:
  - provide an algorithm
  - show that the resource bounds are respected
- Lower bounds (hardness for a class) are usually difficult to show:

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:
  - provide an algorithm
  - show that the resource bounds are respected
- Lower bounds (hardness for a class) are usually difficult to show:
  - the technical tool here is the polynomial reduction (or any other appropriate reduction)

Computational Complexity: Upper and lower bounds

- Upper bounds (membership in a class) are usually easy to prove:
  - provide an algorithm
  - show that the resource bounds are respected
- Lower bounds (hardness for a class) are usually difficult to show:
  - the technical tool here is the polynomial reduction (or any other appropriate reduction)
  - show that some hard problem can be reduced to the problem at hand

Computational Complexity: Polynomial reduction

- Given languages L_1 and L_2, L_1 can be polynomially reduced to L_2, written $L_1 \leq_p L_2$, if there exists a polynomial time-computable function f such that

$$x \in L_1 \iff f(x) \in L_2.$$

Rationale: it cannot be harder to decide L_1 than L_2

Computational Complexity: Polynomial reduction

- Given languages L_1 and L_2, L_1 can be polynomially reduced to L_2, written $L_1 \leq_p L_2$, if there exists a polynomial time-computable function f such that

$$x \in L_1 \iff f(x) \in L_2.$$

  Rationale: it cannot be harder to decide L_1 than L_2
- L is hard for a class C (C-hard) if all languages of this class can be reduced to L.

Computational Complexity: Polynomial reduction

- Given languages L_1 and L_2, L_1 can be polynomially reduced to L_2, written $L_1 \leq_p L_2$, if there exists a polynomial time-computable function f such that

$$x \in L_1 \iff f(x) \in L_2.$$

  Rationale: it cannot be harder to decide L_1 than L_2
- L is hard for a class C (C-hard) if all languages of this class can be reduced to L.
- L is complete for C (C-complete) if L is C-hard and $L \in C$.

Computational Complexity: NP-complete problems

- A problem is NP-complete iff it is NP-hard and in NP.

Computational Complexity: NP-complete problems

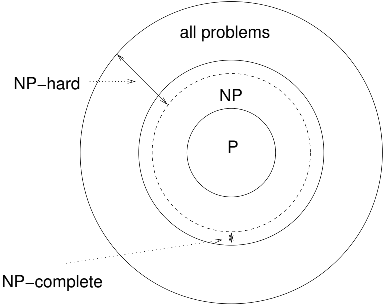
- A problem is NP-complete iff it is NP-hard and in NP.
- Example: SAT (the satisfiability problem for propositional logic) is NP-complete (Cook/Karp)

Computational Complexity: NP-complete problems

- A problem is NP-complete iff it is NP-hard and in NP.
- Example: SAT (the satisfiability problem for propositional logic) is NP-complete (Cook/Karp)
  - Membership is obvious, hardness follows because computations on a NDTM correspond to satisfying truth assignments of certain formulae

Computational Complexity: NP-complete problems

- A problem is NP-complete iff it is NP-hard and in NP.
- Example: SAT (the satisfiability problem for propositional logic) is NP-complete (Cook/Karp)
  - Membership is obvious, hardness follows because computations on a NDTM correspond to satisfying truth assignments of certain formulae



Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!
- Define co-C = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in C\}$ (provided Σ is our alphabet)

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!
- Define co-C = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in C\}$ (provided Σ is our alphabet)
- co-NP = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in \text{NP}\}$

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!
- Define co-C = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in C\}$ (provided Σ is our alphabet)
- co-NP = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in \text{NP}\}$
- Examples: UNSAT, TAUT $\in$ co-NP!

Computational Complexity: The complexity class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - In other words: Checking a proposed solution (of poly size) is easy.
  - What if we want to decide UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!
- Define co-C = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in C\}$ (provided Σ is our alphabet)
- co-NP = $\{L \subseteq \Sigma^* : \Sigma^* \setminus L \in \mathrm{NP}\}$
- Examples: UNSAT, TAUT $\in$ co-NP!
- *Note:* P is closed under complement, in particular,

$$\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$$

Computational Complexity: PSPACE

There are problems even more difficult than NP and co-NP…

Computational Complexity: PSPACE

There are problems even more difficult than NP and co-NP...

Definition ((N)PSPACE)

PSPACE (NPSPACE) is the class of decision problems that can be decided on deterministic (non-deterministic) Turing machines using only polynomially many tape cells.

Computational Complexity: PSPACE

There are problems even more difficult than NP and co-NP…

Definition ((N)PSPACE)

PSPACE (NPSPACE) is the class of decision problems that can be decided on deterministic (non-deterministic) Turing machines using only polynomially many tape cells.

Some facts about PSPACE:

- PSPACE is closed under complements (… as all other deterministic classes)
- PSPACE is identical to NPSPACE (because non-deterministic Turing machines can be simulated on deterministic TMs using only quadratic space: Savitch's Theorem)
- NP$\subseteq$PSPACE (because in polynomial time one can "visit" only polynomial space, i.e., NP$\subseteq$NPSPACE)

Computational Complexity: PSPACE-completeness

Definition (PSPACE-completeness)

A decision problem (or language) is PSPACE-complete if it is in PSPACE and all other problems in PSPACE can be polynomially reduced to it.

Computational Complexity: PSPACE-completeness

Definition (PSPACE-completeness)

A decision problem (or language) is PSPACE-complete if it is in PSPACE and all other problems in PSPACE can be polynomially reduced to it.

Intuitively, PSPACE-complete problems are the “hardest” problems in PSPACE (similar to NP-completeness). They appear to be “harder” than NP-complete problems from a practical point of view.

Computational Complexity: PSPACE-completeness

Definition (PSPACE-completeness)

A decision problem (or language) is PSPACE-complete if it is in PSPACE and all other problems in PSPACE can be polynomially reduced to it.

Intuitively, PSPACE-complete problems are the “hardest” problems in PSPACE (similar to NP-completeness). They appear to be “harder” than NP-complete problems from a practical point of view.

An example for a PSPACE-complete problem is the NDFA equivalence problem:

Instance: Two non-deterministic finite state automata A_1 and A_2.

Question: Are the languages accepted by A_1 and A_2 identical?

Computational complexity of MAPF/DU bounded plan existence

Theorem

Deciding whether there exists an eager MAPF/DU i-strong or objectively strong plan with execution cost k or less is PSPACE-complete.

Proof sketch.

Since plans have polynomial depth, all execution traces can be generated non-deterministically and tested using only polynomial space, i.e., PSPACE-membership.

Computational complexity of MAPF/DU bounded plan existence

Theorem

Deciding whether there exists an eager MAPF/DU i-strong or objectively strong plan with execution cost k or less is PSPACE-complete.

Proof sketch.

Since plans have polynomial depth, all execution traces can be generated non-deterministically and tested using only polynomial space, i.e., PSPACE-membership. For hardness, reduction from QBF.

Computational complexity of MAPF/DU bounded plan existence

Theorem

Deciding whether there exists an eager MAPF/DU i-strong or objectively strong plan with execution cost k or less is PSPACE-complete.

Proof sketch.

Since plans have polynomial depth, all execution traces can be generated non-deterministically and tested using only polynomial space, i.e., PSPACE-membership.
For hardness, reduction from QBF.
Example construction for

$\forall x_1 \exists x_2 \forall x_3$:

$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$

Computational complexity of MAPF/DU bounded plan existence

Theorem

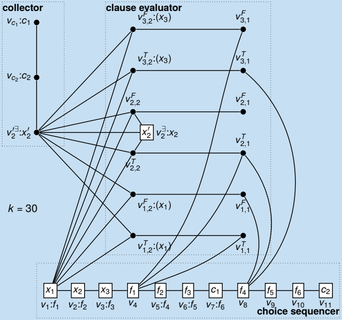
Deciding whether there exists an eager MAPF/DU i-strong or objectively strong plan with execution cost k or less is PSPACE-complete.

Proof sketch.

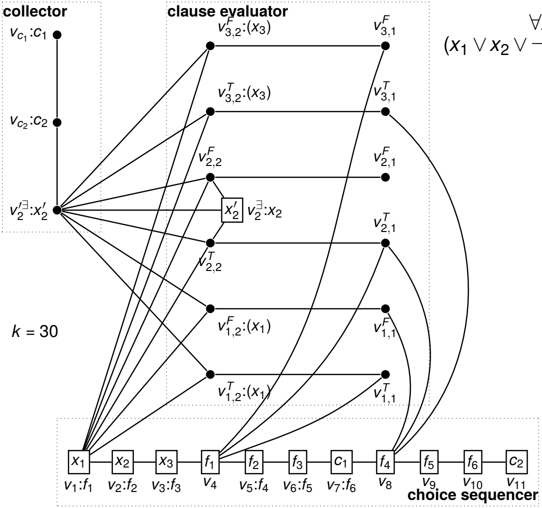
Since plans have polynomial depth, all execution traces can be generated non-deterministically and tested using only polynomial space, i.e., PSPACE-membership. For hardness, reduction from QBF. Example construction for

$\forall x_1 \exists x_2 \forall x_3 :$

$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$



The reduction enlarged



$$\forall x_1 \exists x_2 \forall x_3 :$$
$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

Complexity with a fixed number of agents

These results probably imply that the technique could not be used online.

For a fixed number of agents, however, the bounded planning problem is polynomial.

Theorem

For a fixed number c of agents, deciding whether there exists a MAPF/DU i-strong or objectively strong plan with execution cost of k or less can be done in time $O(n^{c^2+c})$.

That means, for two agents, it takes “only” $O(n^6)$ time – but in practice it should be faster.

An algorithm for generating an objective MAPF/DU plan for two agents

1. Determine in the *state space of all node assignments* the distance to the initial state using Dijkstra: $O(|V|^4)$ time.

An algorithm for generating an objective MAPF/DU plan for two agents

1. Determine in the *state space of all node assignments* the distance to the initial state using Dijkstra: $O(|V|^4)$ time.
2. For each of the $O(|V|^2)$ configurations check, whether it is a *potential stepping stone* for one agent, i.e., whether all potential destinations of this agent are reachable using Dijkstra on the modified graph, where the other agent blocks the way: $O(|V|^4)$ time.

An algorithm for generating an objective MAPF/DU plan for two agents

UNI FREIBURG

1. Determine in the *state space of all node assignments* the distance to the initial state using Dijkstra: $O(|V|^4)$ time.
2. For each of the $O(|V|^2)$ configurations check, whether it is a *potential stepping stone* for one agent, i.e., whether all potential destinations of this agent are reachable using Dijkstra on the modified graph, where the other agent blocks the way: $O(|V|^4)$ time.
3. For all $O(|V|^2)$ potential stepping stones, check whether for each of the $O(|V|)$ possible destination of the first agent, the second agent can reach its possible destinations and use Dijkstra to compute the shortest path: altogether $O(|V|^5)$ time.

An algorithm for generating an objective MAPF/DU plan for two agents

1. Determine in the *state space of all node assignments* the distance to the initial state using Dijkstra: $O(|V|^4)$ time.
2. For each of the $O(|V|^2)$ configurations check, whether it is a *potential stepping stone* for one agent, i.e., whether all potential destinations of this agent are reachable using Dijkstra on the modified graph, where the other agent blocks the way: $O(|V|^4)$ time.
3. For all $O(|V|^2)$ potential stepping stones, check whether for each of the $O(|V|)$ possible destination of the first agent, the second agent can reach its possible destinations and use Dijkstra to compute the shortest path: altogether $O(|V|^5)$ time.
4. Consider all stepping stones and minimize over the maximum plan depth. Among the minimal plans select those that are eager for the planning agent.

Summary

- DMAPF generalizes the MAPF problem by dropping the assumption that plans are generated centrally and then communicated.
- MAPF/DU generalizes the MAPF problem further by dropping the assumptions that destinations are common knowledge.
- A solution concept for this setting are *i*-strong branching plans corresponding to implicitly coordinated policies in the area of epistemic planning.
- The backbone of such plans are stepping stones.
- Joint execution can be guaranteed to be successful and polynomially bounded if all agents are conservative and optimally eager.
- While bounded plan existence in general is PSPACE-complete, it is polynomial for a fixed number of

Outlook

- → Do the results still hold for planar graphs?
- Is MAPF/DU plan existence also PSPACE-complete?
- How would more general forms of describing the common knowledge about destinations affect the results?
- → Overlap of destinations or general Boolean combinations
- Can we get similar results for other execution semantics?
- → Concurrent executions of actions
- Can we be more aggressive in expectations about possible destinations?
- → Use forward induction, i.e., assume that actions in the past were rational.
- Are other forms of implicit coordination possible?
- → More communication? Coordination in competitive scenarios?

Literature

B. Nebel, T. Bolander, T. Engesser and R. Mattmüller.
Implicitly Coordinated Multi-Agent Path Finding under Destination Uncertainty.
Journal of Artificial Intelligence research 64: 497-527, 2019.

T. Bolander, T. Engesser, R. Mattmüller and B. Nebel.
Better Eager Than Lazy? How Agent Types Impact the Successfulness of Implicit Coordination.
In *Proceedings of the Sixteenth Conference on Principles of Knowledge Representation and Reasoning (KR-18)*, pages 445-453. 2018.

T. Engesser, T. Bolander, R. Mattmüller, and B. Nebel.
Cooperative epistemic multi-agent planning for implicit coordination.
In *Proceedings of the Ninth Workshop on Methods for Modalities (M4MICLA-17)*, pages 75–90, 2017.