Multi-agent Path Finding Planning & Executing

Roman Barták
Charles University, Czech Republic
joint work **Jiří Švancara** and **Ivan Krasičenko**

Part I: Introduction to MAPF

– *Problem formulation, variants and objectives*

Part II. Solving MAPF

– *Reduction-based solvers*

Part III. From abstract to executable actions

– *Translation vs. model modification*

Part IV. Demo

Part I: Introduction to MAPF

– *Problem formulation, variants and objectives*

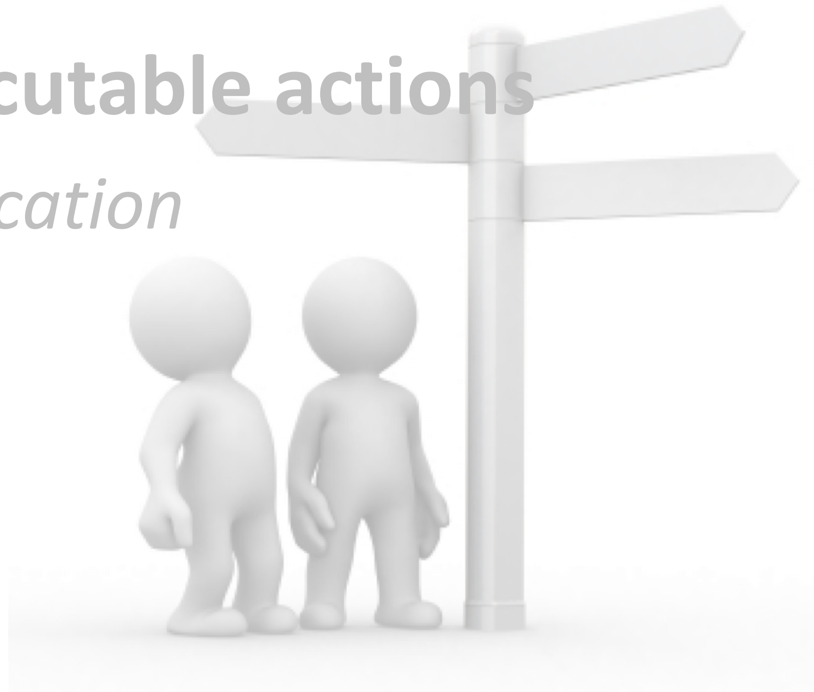
Part II. Solving MAPF

– *Reduction-based solvers*

Part III. From abstract to executable actions

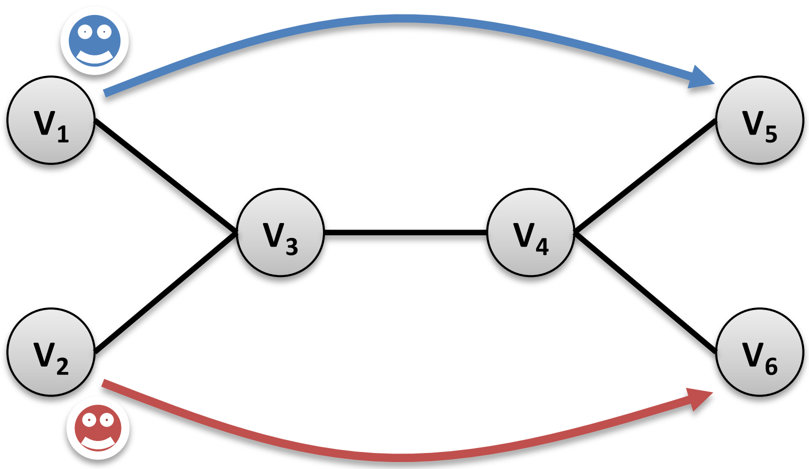
– *Translation vs. model modification*

Part IV. Demo



- an (undirected) **graph**
- a set of **agents**, each agent is assigned to two locations (nodes) in the graph (start, destination)
- agents can **move** (to a neighboring node) or **wait**

Find **plans** for all agents such that the plans **do not collide in time and space** (no two agents are at the same location at the same time).



time	agent 1	agent 2
0	$\mathbf{v_1}$	$\mathbf{v_2}$
1	wait $\mathbf{v_1}$	move $\mathbf{v_3}$
2	move $\mathbf{v_3}$	move $\mathbf{v_4}$
3	move $\mathbf{v_4}$	move $\mathbf{v_6}$
4	move $\mathbf{v_5}$	wait $\mathbf{v_6}$

Vertex conflict – two agents are at the same time at the same vertex

Edge conflict – two agents use the same edge at the same direction

Swapping conflict – two agents use the same edge at different direction

Following conflict – one agent follows another one (train)

Cycle conflict – agents are following each other forming a “rotating cycle” pattern

How to measure quality of plans?

Two typical criteria (to minimize):

- **Makespan**
  - distance between the start time of the first agent and the completion time of the last agent
  - maximum of lengths of plans (end times)
- **Sum of costs (SOC)**
  - sum of lengths of plans (end times)

time	agent 1	agent 2
0	**v_1**	**v_2**
1	wait **v_1**	move **v_3**
2	move **v_3**	move **v_4**
3	move **v_4**	move **v_6**
4	move **v_5**	wait **v_6**

Makespan = 4
SOC = 7

Part I: Introduction to MAPF

– *Problem formulation, variants and objectives*

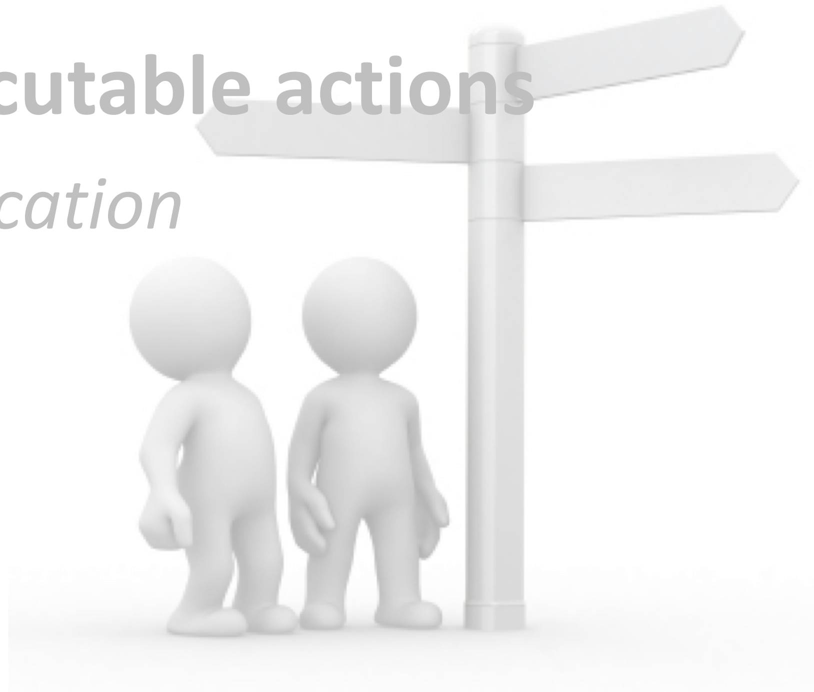
Part II. Solving MAPF

– *Reduction-based solvers*

Part III. From abstract to executable actions

– *Translation vs. model modification*

Part IV. Demo



Express (model) the problem as a **SAT formula** in a conjunctive normal form (CNF)

- Boolean *variables* (true/false values)
- *clause* = a disjunction of literals (variables and negated variables)
- *formula* = a conjunction of clauses
- *solution* = an instantiation of variables such that the formula is satisfied

Example:

- (X *or* Y) *and* (*not* X *or not* Y)
- [exactly one of X and Y is true]

SAT model is expressed as a CNF formula

We can go beyond CNF and use **abstract expressions** that are translated to CNF.

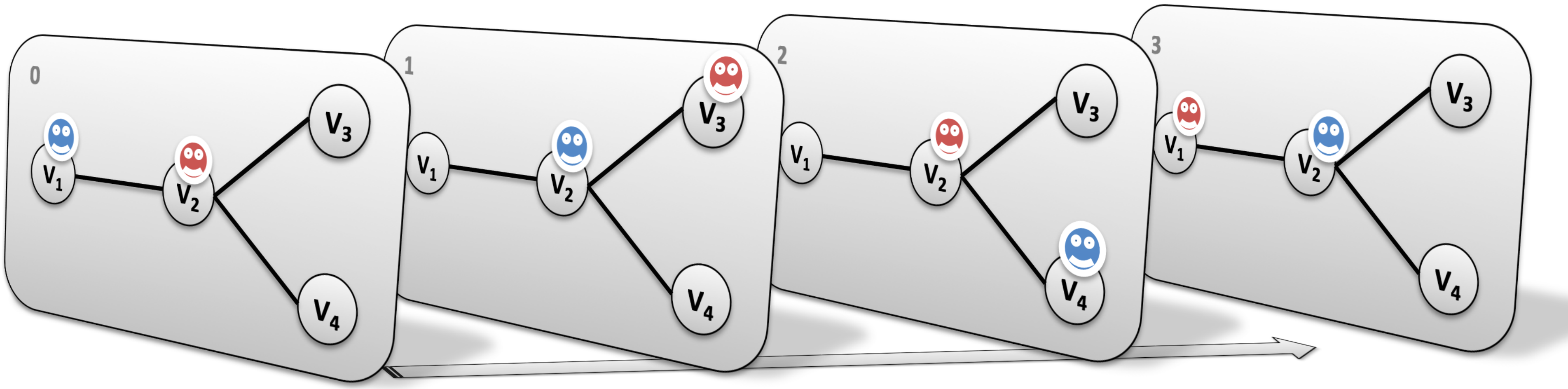
A => B	B *or not* A
sum(Bs) >= 1 (at-least-one(Bs))	disj(Bs)
sum(Bs) = 1	at-most-one(Bs) *and* at-least-one(Bs)

We can even use **numerical variables** (and constraints).

In MAPF, we do not know the lengths of plans (due to possible re-visits of nodes)!

We can encode plans of a known length using a **layered graph** (temporally extended graph).

- Each layer corresponds to one time slice and indicates positions of agents at that time.



Using **layered graph** describing agent positions at each time step

B_{tav} : agent *a* occupies vertex *v* at time *t*

Constraints:

- each agent occupies exactly one vertex at each time.

$$\Sigma_{v=1}^{n} B_{tav} = 1 \text{ for } t = 0, \ldots, m, \text{ and } a = 1, \ldots, k.$$

- no two agents occupy the same vertex at any time.

$$\Sigma_{a=1}^{k} B_{tav} \leq 1 \text{ for } t = 0, \ldots, m, \text{ and } v = 1, \ldots, n.$$

- if agent *a* occupies vertex *v* at time *t*, then *a* occupies a neighboring vertex or stay at *v* at time *t* + 1.

$$B_{tav} = 1 \Rightarrow \Sigma_{u \in neibs(v)} (B_{(t+1)au}) \geq 1$$

Preprocessing:

B_{tav} = 0 if agent *a* cannot reach vertex *v* at time *t* or *a* cannot reach the destination being at *v* at time *t*

Picat code

```
import sat.

path(N,As) =>
    K = len(As),
    lower_upper_bounds(As,LB,UB),
    between(LB,UB,M),
    B = new_array(M+1,K,N),
    B :: 0..1,

    % Initialize the first and last states
    foreach (A in 1..K)
        (V,FV) = As[A],
        B[1,A,V] = 1,
        B[M+1,A,FV] = 1
    end,

    % Each agent occupies exactly one vertex
    foreach (T in 1..M+1, A in 1..K)
        sum([B[T,A,V] : V in 1..N]) #= 1
    end,

    % No two agents occupy the same vertex
    foreach (T in 1..M+1, V in 1..N)
        sum([B[T,A,V] : A in 1..K]) #=< 1
    end,

    % Every transition is valid
    foreach (T in 1..M, A in 1..K, V in 1..N)
        neibs(V,Neibs),
        B[T,A,V] #=>
        sum([B[T+1,A,U] : U in Neibs]) #>= 1
    end,

    solve(B),
    output_plan(B).
```

Incremental generation of layers

Setting the initial and destination locations

Agent occupies one vertex at any time

No conflict between agents

Agent moves to a neighboring vertex

```
foreach(T in 1..M1, A in 1..K, V in 1..N)
    B[T,A,V] #=> sum([B[Prev,A2,V] :
                 A2 in 1..K, A2!=A,
                 Prev in max(1,T-L)..T]) #= 0
end
```

K-robustness

SAT encoding

At(x,a,t) – agent **a** is at node **x** at time **t**
Pass(x,y,a,t) – agent **a** is going from node **x** to node **y** at time **t**

$$\forall a \in A : At(s_a, a, 0) = 1$$

- initial location

$$\forall a \in A : At(g_a, a, T) = 1$$

- goal location

$$\forall a \in A, \forall t \in \{0, \ldots, T\} : \sum_{x \in V} At(x, a, t) \leq 1$$

- at most one node per agent

$$\forall x \in V, \forall t \in \{0, \ldots, T\} : \sum_{a \in A} At(x, a, t) \leq 1$$

- at most one agent per node (no vertex conflict)

$$\forall x \in V, \forall a \in A, \forall t \in \{0, \ldots, T-1\} : \quad At(x, a, t) \implies \sum_{(x,y) \in E} Pass(x, y, a, t) = 1$$

- from node to edge

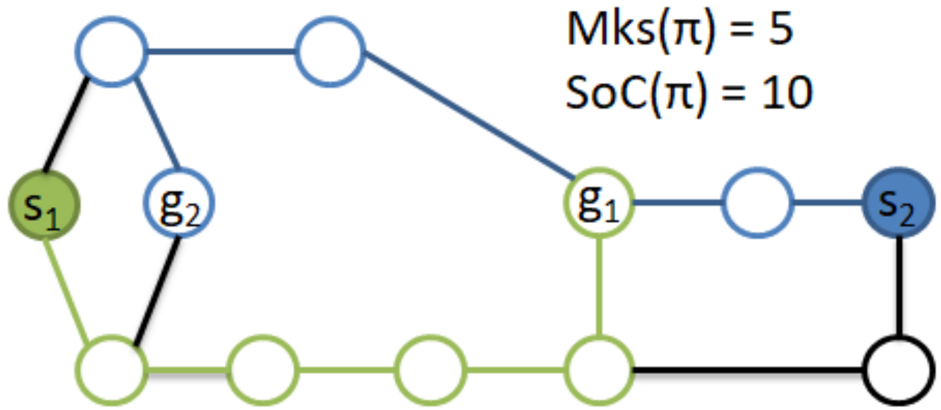
$$\forall (x, y) \in E, \forall a \in A, \forall t \in \{0, \ldots, T-1\} : \quad Pass(x, y, a, t) \implies At(y, a, t+1)$$

- from edge to node

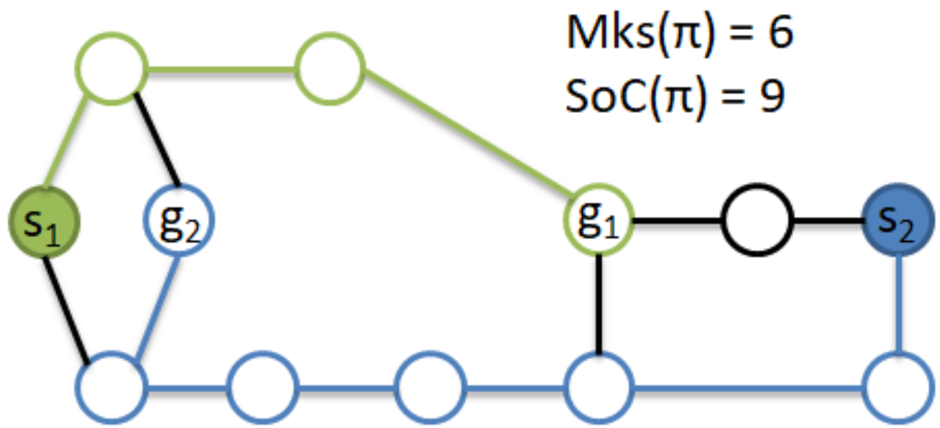
$$\forall (x, y) \in E : x \neq y, \forall t \in \{0, \ldots, T-1\} : \quad \sum_{a \in A} (Pass(x, y, a, t) + Pass(y, x, a, t)) \leq 1$$

- no swapping conflict

Makespan-optimal plan might be SOC-suboptimal



and vice versa, SOC-optimal plan may require larger makespan.



SOC – Model 1 (incremental)

Calculate shortest plan for each agent independently

Algorithm 1 Model 1

function MODEL 1
  $\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$
  $LB(Mks) = \max_{i \in A} SP_i$
  $LB(SoC) = \sum_{i \in A} SP_i$
  $\delta \leftarrow 0$
  while No Solution **do**
    solve_MAPF($LB(Mks) + \delta, LB(SoC) + \delta$)
    $\delta \leftarrow \delta + 1$
  end while
end function

Calculate lower bounds for makespan and SOC

Look for a plan with this makespan and with upper bound for SOC

If the plan exists then we are done

If the plan does not exist then add one time layer and increase upper bound for SOC

Observation:

- When we finally find the SOC-optinal plan, we noticed that a smaller makespan would be enough in many cases (but when this makespan was explored, the upper bound for SOC was too tight).

Core idea:

- Find a plan with minimal makespan and use the difference between SOC of that plan and the lower bound for SOC to find how many extra time layers are needed.

Algorithm 2 Model 2

function MODEL 2
  $\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$
  $LB(Mks) = \max_{i \in A} SP_i$
  $LB(SoC) = \sum_{i \in A} SP_i$
  $\gamma \leftarrow 0$
  while No Solution **do**
    $SoC \leftarrow$ opt_MAPF$(LB(Mks) + \gamma, LB(SoC), |A| * LB(Mks) + \gamma)$
    $\gamma \leftarrow \gamma + 1$
  end while
  $\delta \leftarrow SoC - LB(SoC)$
  opt_MAPF$(LB(Mks) + \delta, LB(SoC), SoC)$
end function

Calculate shortest plan for each agent independently

Calculate lower bounds for makespan and SOC

Look for a plan with the minimal makespan and for that makespan find the best SOC plan

Calculate the needed makespan and find best SOC plan for it

Classical pre-processing

- node x is not reachable from the start node at time t (or destination is not reachable from node x when starting at time t)

  => *At(x,a,t) = 0*

Novel pre-processing (for SOC)

- Let Sp_a be length of the shortest path for agent a, minSOC be the lower bound for SOC, and minSOC+d be the current upper-bound for SOC

  $\Rightarrow$ agent a must be at its destination since time Sp_i+d

  $\Rightarrow$ *At(x,a,t) = 0* ($x \neq g_a$ & $t \geq Sp_a$ +d)

4-connected grid maps (8x8 to 16x16)

- 20% randomly placed obstacles

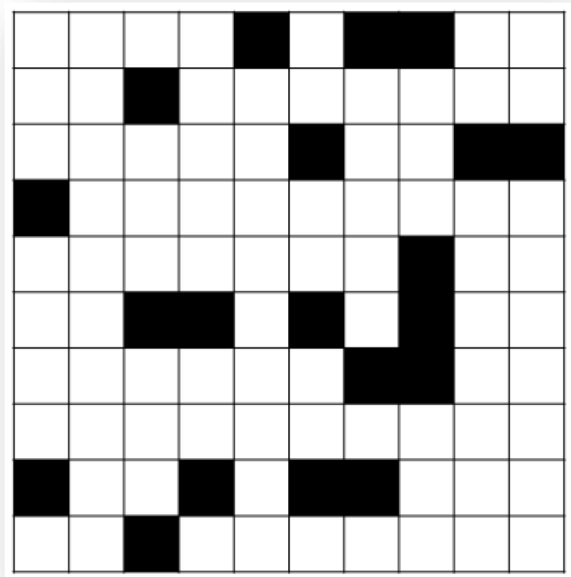
for grid WxW, we use W to 2W agents

- randomly placed starts/goals

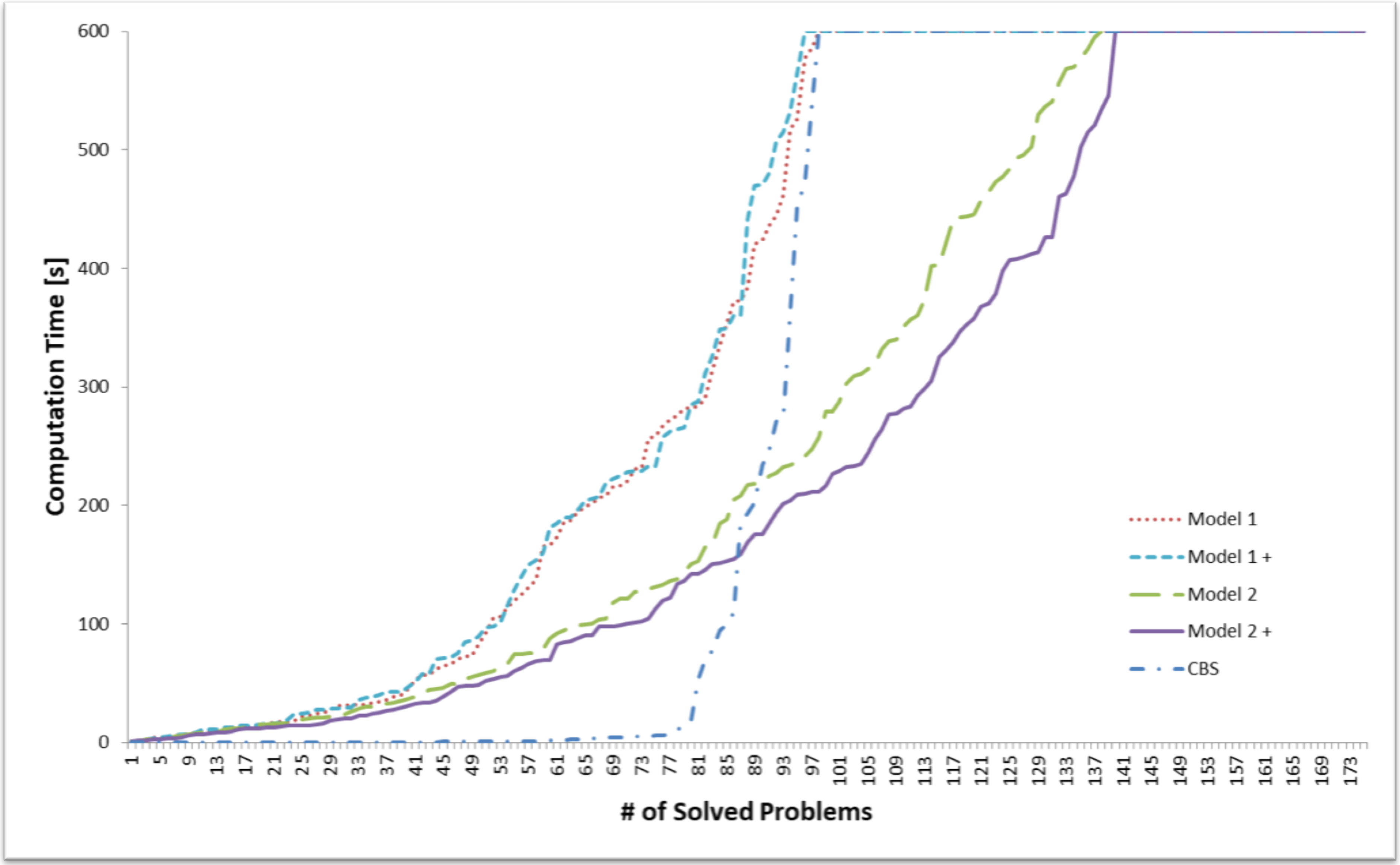
five instances for each setting

175 unique problem instances

time limit of 600 seconds



Results



Computation Time [s]
600
500
400
300
200
100
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117 121 125 129 133 137 141 145 149 153 157 161 165 169 173
# of Solved Problems
Model 1
Model 1 +
Model 2
Model 2 +
CBS

Results (another perspective)

	M. 1	M. 2	M. 1+	M. 2+	CBS
# of solved	97	137	95	**139**	97
# of fastest	0	4	3	46	**88**
# of fastest (without CBS)	9	8	6	**118**	–
IPC score	16.11	44.56	16.76	57.07	**92.50**
IPC score (without CBS)	57.19	110.54	53.93	**134.29**	–

Part I: Introduction to MAPF

- *Problem formulation, variants and objectives*

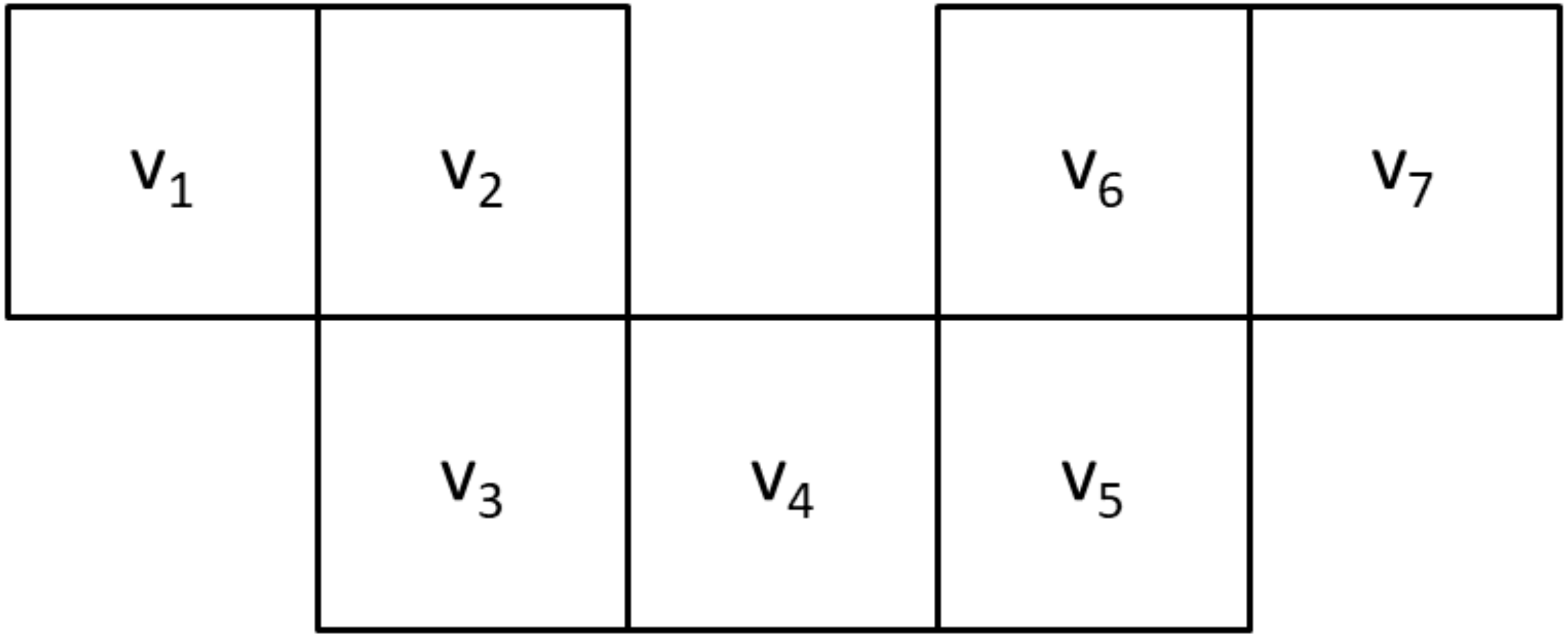
Part II. Solving MAPF

- *Reduction-based solvers*

Part III. From abstract to executable actions

- *Translation vs. model modification*

Part IV. Demo



6 classical actions needed to go from v1 to v7

plus 4 turning actions during execution

turning may take significant time (w.r.t. moving)

Abstract actions:

- move
- wait

Executable actions:

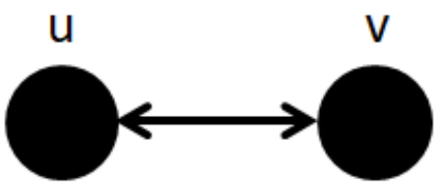
- move forward
- wait
- turn left/right + move
- turn back and move

Times:

t_t – time to turn left/right
t_f – time to move forward

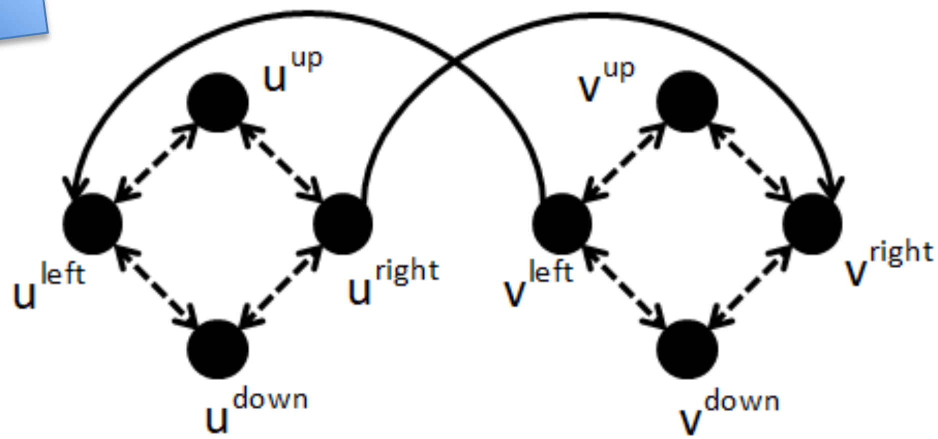
classic	classic+wait
t_f	$t_f + 2*t_t$
$t_f + t_t/2$	$t_f + 2*t_t$
$t_f + t_t$	$t_f + 2*t_t$
$t_f + 2*t_t$	$t_f + 2*t_t$

It is possible to assume turn actions during path finding by splitting the nodes.

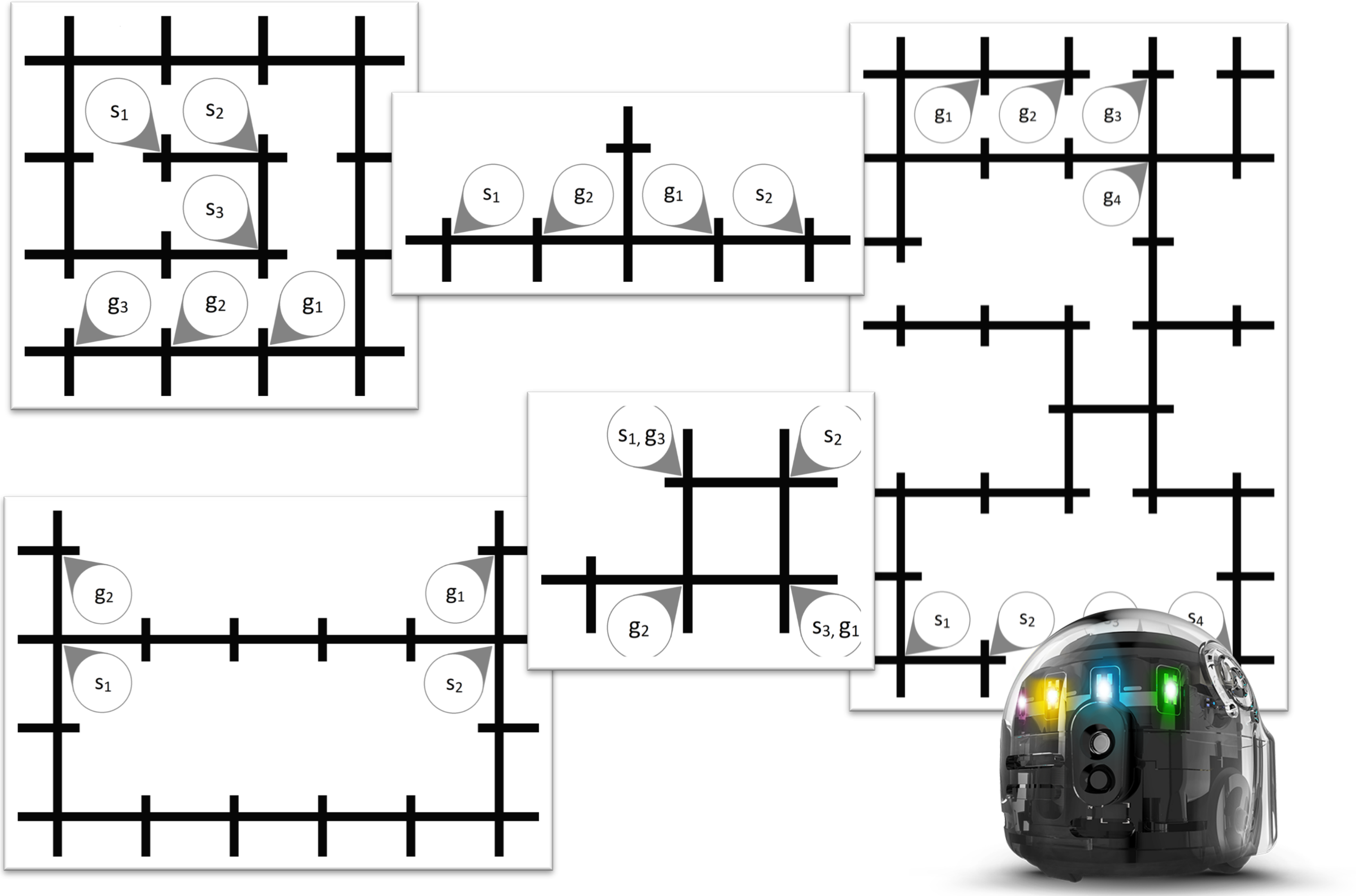


Classical model

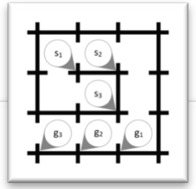
Split model



Experiment setting



Some results



edge 5 cm

10 cm

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	14	14	5	0	1	0	NA	49.2	1.6	1.6
classic+wait	14	14	0	0	6	0	43.8	64.3	0	0
classic+robustness	16	16	0	0	0	0	32.7	56.3	1.7	1.5
classic+wait+robustness	16	16	0	0	0	0	50.1	74	0	0
split	22	22	0	0	0	0	30.3	52.3	1.3	2.3
split+wait	22	22	0	0	6	0	36.1	69.1	0	0
split+robustness	23	23	0	0	0	0	31.2	53.1	1.2	2.2
split+wait+robustness	23	23	0	0	0	0	37.5	72.2	0	0
w-split	36	66	0	0	0	0	30.2	54	0	0
w-split+robustness	36	66	0	0	0	0	30.2	54.1	0	0

Quality index	Computed Makespan		Failed Runs		Number of Collisions		Total Time		Max Δ time	
classic	**5.00**	**5.00**	2.00	**5.00**	2.75	**5.00**	1.90	**4.93**	1.52	1.61
classic+wait	**5.00**	**5.00**	**5.00**	**5.00**	2.12	**5.00**	3.69	4.10	**5.00**	**5.00**
classic+robustness	3.95	3.95	**5.00**	**5.00**	**5.00**	**5.00**	4.12	3.98	2.64	2.74
classic+wait+robustness	3.95	3.95	**5.00**	**5.00**	**5.00**	**5.00**	2.79	3.08	**5.00**	**5.00**
split	3.04	3.04	**5.00**	4.17	3.70	4.00	4.80	3.72	2.35	1.82
split+wait	3.04	3.04	**5.00**	**5.00**	2.73	**5.00**	4.11	3.50	**5.00**	**5.00**
split+robustness	2.87	2.87	4.17	4.17	4.33	4.50	3.67	3.57	3.14	2.69
split+wait+robustness	2.87	2.87	**5.00**	**5.00**	**5.00**	**5.00**	3.83	3.29	**5.00**	**5.00**
w-split	1.97	1.15	**5.00**	**5.00**	3.83	**5.00**	**4.99**	4.88	**5.00**	**5.00**
w-split+robustness	1.92	1.13	**5.00**	**5.00**	**5.00**	**5.00**	4.88	4.82	**5.00**	**5.00**

Part I: Introduction to MAPF

– *Problem formulation, variants and objectives*

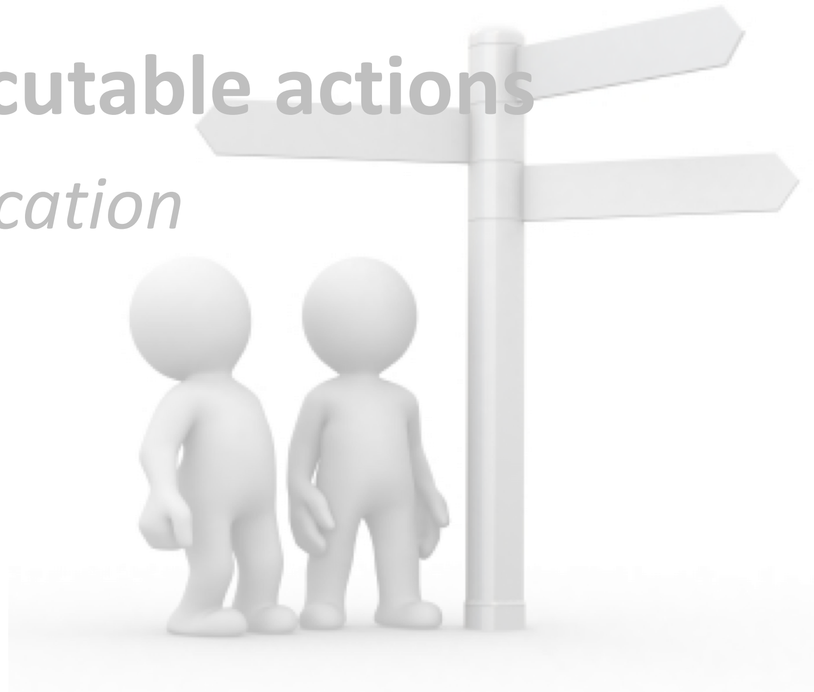
Part II. Solving MAPF

– *Reduction-based solvers*

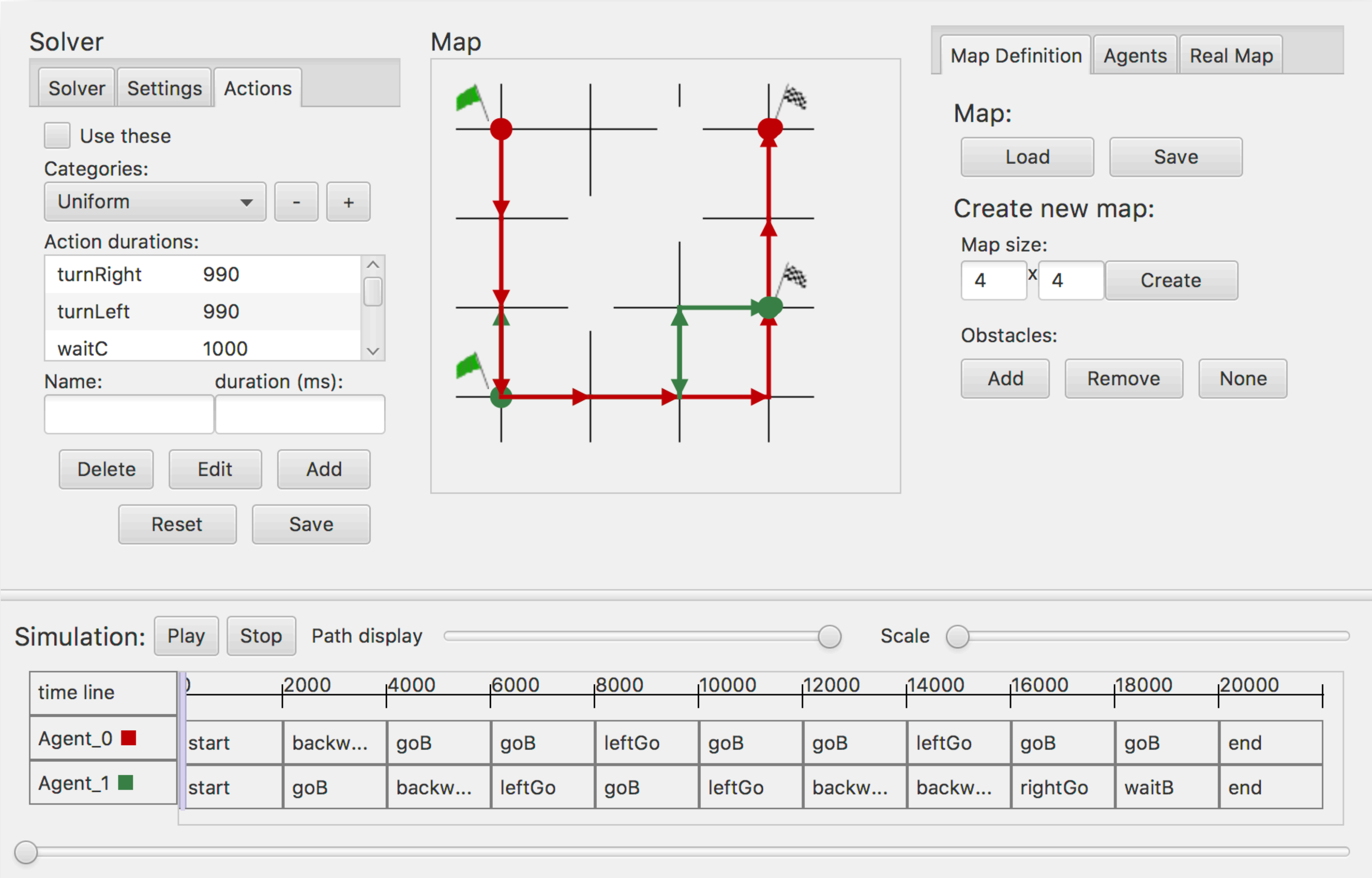
Part III. From abstract to executable actions

– *Translation vs. model modification*

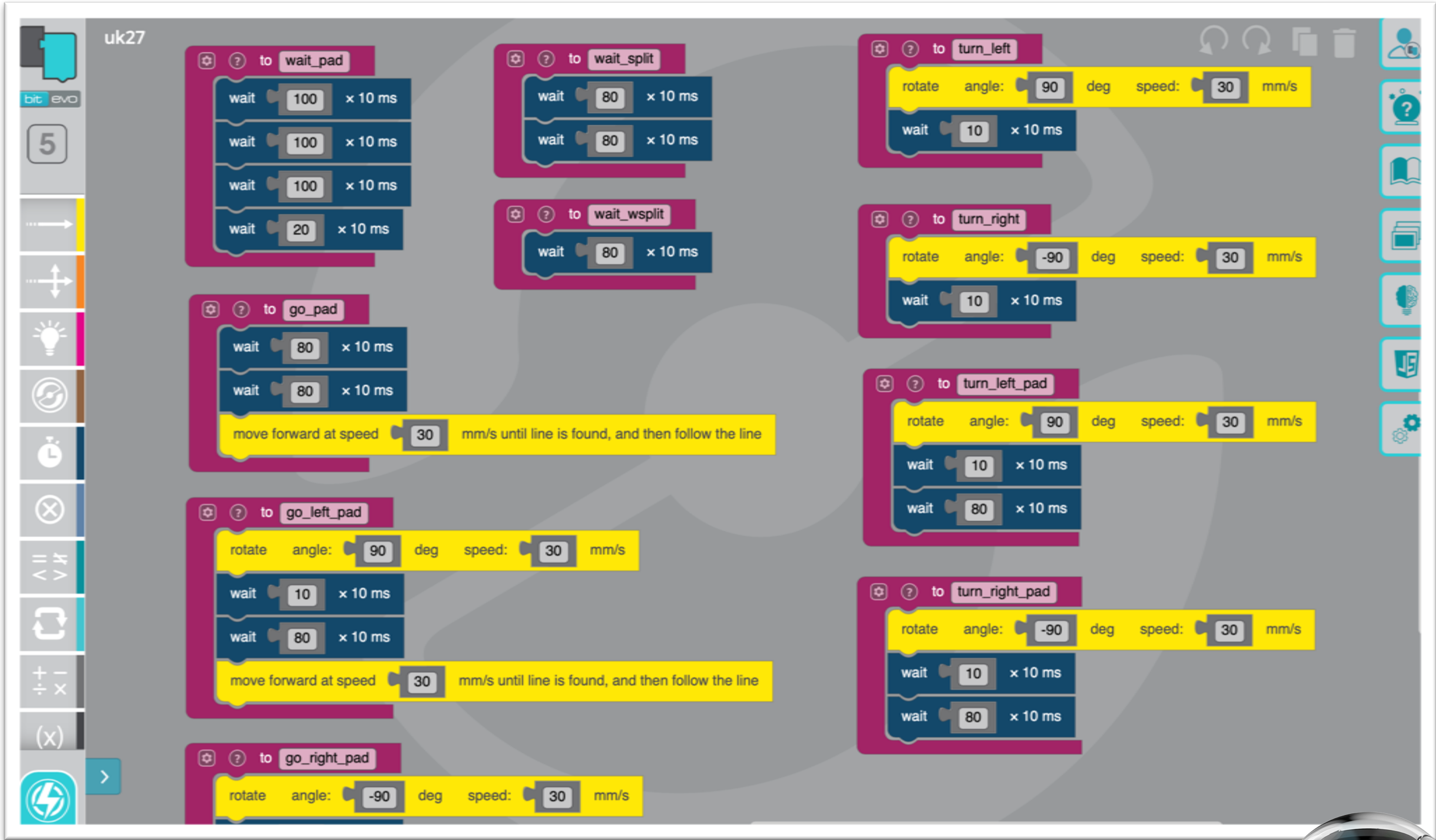
Part IV. Demo



MAPF software



Create a map


uk27
bit evo
5
to wait_pad
wait 100 × 10 ms
wait 100 × 10 ms
wait 100 × 10 ms
wait 20 × 10 ms
to wait_split
wait 80 × 10 ms
wait 80 × 10 ms
to wait_wsplit
wait 80 × 10 ms
to go_pad
wait 80 × 10 ms
wait 80 × 10 ms
move forward at speed 30 mm/s until line is found, and then follow the line
to go_left_pad
rotate angle: 90 deg speed: 30 mm/s
wait 10 × 10 ms
wait 80 × 10 ms
move forward at speed 30 mm/s until line is found, and then follow the line
to go_right_pad
rotate angle: -90 deg speed: 30 mm/s
to turn_left
rotate angle: 90 deg speed: 30 mm/s
wait 10 × 10 ms
to turn_right
rotate angle: -90 deg speed: 30 mm/s
wait 10 × 10 ms
to turn_left_pad
rotate angle: 90 deg speed: 30 mm/s
wait 10 × 10 ms
wait 80 × 10 ms
to turn_right_pad
rotate angle: -90 deg speed: 30 mm/s
wait 10 × 10 ms
wait 80 × 10 ms

Roman Barták
Charles University, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz