INTRODUCTION TO AI
STRIPS PLANNING

.. and Applications to Video-games!
Lecture 1: Game-inspired competitions for AI research, AI decision making for non-player characters in games

Lecture 2: STRIPS planning, state-space search

Lecture 3: Planning Domain Definition Language (PDDL), using an award winning planner to solve Sokoban

Lecture 4: Planning graphs, domain independent heuristics for STRIPS planning

Lecture 5: Employing STRIPS planning in games: SimpleFPS, iThinkUnity3D, SmartWorkersRTS

Lecture 6: Planning beyond STRIPS
Planning graphs

- Planning graph
Planning graphs

- Planning graph
  - Special data structure

- Easy to compute: polynomial complexity!

- Can be used by the GRAPHPLAN algorithm to search for a solution (following similar reasoning as in the example)

- Can be used as a guideline for heuristic functions for progressive planning that are more accurate than the ones we sketched in Lecture 1
Planning graphs

- Planning graph
  - Special data structure
  - Easy to compute: polynomial complexity!
  - Can be used by the GRAPHPLAN algorithm to search for a solution (following similar reasoning as in the example)
  - Can be used as a guideline for heuristic functions for progressive planning that are more accurate than the ones we sketched in Lecture 1
Planning graphs

- Planning graph
  - Computing the graph has **polynomial** complexity

- STRIPS planning
  - Finding a solution is **PSPACE-complete**

- Where’s the complexity hiding?
Planning graphs

- Planning graph
  - Computing the graph has polynomial complexity

- Finding a solution using the graph is NP-complete, while we may also need to extend the graph a finite number of times… \( \rightarrow \) PSPACE
Planning graphs

- Planning graph
  - Special data structure
    - Easy to compute: polynomial complexity!
    - Can be used by the GRAPHPLAN algorithm to search for a solution (following similar reasoning as in the example)
    - Can be used as a guideline for heuristic functions for progressive planning that are more accurate than the ones we sketched in Lecture 2
Planning graphs

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state
- Compute the successor states
- Pick the most promising of the successor states as the current state
- Repeat until a solution is found or the state space is exhausted
Planning graphs

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state

Compute a planning graph for each successor state to estimate goal distance

- Repeat until a solution is found or the state space is exhausted
Heuristic functions based on planning graphs

- **Level cost**: the level where a literal appears in the graph for the first time
  - Note: A literal that does not appear in the final level of the graph cannot be achieved by any plan!

- **Max-level**: the max of the level cost for each sub-goal
- **Sum-level**: the sum of the level cost for each sub-goal
- **Set-level**: the first level that all sub-goals appear together without mutexes
Planning graphs

As an example let’s see the heuristics for the planning graph from the initial state

- Compute applicable actions to the current state
- Compute the successor states
- Pick one the most promising of the successor states as the current state
- Repeat until a solution is found or the state space is exhausted
Planning graphs

- Level cost for sub-goal Have(C) = 0
- Level cost for sub-goal Eaten(C) = 1
- Sum/Max-level heuristic = 1
- Level cost for sub-goal Have(C) = 0
- Level cost for sub-goal Eaten(C) = 1
- Set-level heuristic = 2
Planning graphs

- Heuristic functions based on planning graphs

- As building the planning graph is relatively cheap (polynomial) we can build one for every state we want to evaluate and use Sum/Max/Set-level to estimate the distance to the goal.

- As long as the heuristic provides good estimates, the time spent to calculate the planning graphs pays off because it helps us bypass big parts of the search space.
Planning graphs

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state
- Compute the successor states
- Pick one the most promising of the successor states as the current state
- Repeat until a solution is found or the state space is exhausted
Planning graphs

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state
- Repeat until a solution is found or the state space is exhausted

Here: computing 9 PGs may have helped search a state-space of 1000s of nodes
Relaxed planning task

- Let’s look closer now to one idea we discussed briefly in Lecture 2

- Same as we did with planning graphs, but instead solve a relaxed (i.e., simpler) planning task in order to estimate the goal distance

- Relaxation: Assume an empty list of preconditions
Relaxed planning task

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state
- Compute the successor states
- Pick one the most promising of the successor states as the current state
- Repeat until a solution is found or the state space is exhausted
Relaxed planning task

- Planning graph
  - Computing the graph has **polynomial** complexity

- Empty list of preconditions
  - Finding a solution to the relaxed planning task is **polynomial**
  - OK, but not very informative
Empty list of preconditions

- Initial state
- Goal

Without preconditions you can move each block to the desired position in one step: push(block, from, to, dir)

From every state the goal is at most three actions away
Relaxed planning task

- Let’s look closer now to one idea we discussed briefly in Lecture 1.

- Same as we did with planning graphs, but instead solve a relaxed (i.e., simpler) planning task in order to estimate the goal distance.

- Relaxation: Assume an empty list of negative effects.
Relaxed planning task

- Start from the initial state
- Check if the current state satisfies the goal
- Compute applicable actions to the current state
- Compute the successor states
- Pick one the most promising of the successor states as the current state
- Repeat until a solution is found or the state space is exhausted
Relaxed planning task

- Planning graph
  - Computing the graph has \textbf{polynomial} complexity

- Empty list of negative effects
  - Finding a solution to the relaxed planning task is \textbf{NP-complete}

- It’s not helping...
Relaxed planning task

- Planning graph
  - Computing the graph has **polynomial** complexity

- Empty list of negative effects
  - Finding a solution to the relaxed planning task is **NP-complete**

- We can estimate it!
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- Build a graph that approximates the cost of achieving literal $p$ from state $s$ [Bonet, Geffner 2001]

  - Initialize the graph with literals in $s$ having cost 0
  - For every action $a$ such that $p$ is a **positive effect**, add $p$ and set the cost of $p$ by combining the cost of achieving the preconditions of $a$
  - Build the graph iteratively keeping the minimum cost when a literal $p$ re-appears

- The way the cost is combined for two literals defines the heuristic: $h_{\text{add}}, h_{\text{max}}$
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- Initialize the graph with literals in $s$ having cost 0

$P1: 0$
$P2: 0$
$P3: 0$
$P4: 0$

$s_0$
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- For every action $a$ such that $p$ is a **positive** effect, add $p$ and set the cost of $p$ by combining the cost of achieving the preconditions of $a$.
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- For every action $a$ such that $p$ is a **positive** effect, add $p$ and set the cost of $p$ by combining the cost of achieving the preconditions of $a$.

Additive heuristic $h_{\text{add}}$: sum the cost of preconditions +1

- P1: 0
- P2: 0
- P3: 0
- P4: 0

Add: $P5: (0+0)+1=1$

Add: $P6: (0+0)+1=1$
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- For every action $a$ such that $p$ is a \textbf{positive} effect, add $p$ and set the cost of $p$ by combining the cost of achieving the preconditions of $a$.

Additive heuristic $h_{\text{add}}$: sum the cost of preconditions +1

- $P1: 0$
- $P2: 0$
- $P3: 0$
- $P4: 0$

- $A1$
- $P5: (0+0)+1=1$

- $A2$
- $P6: (0+0)+1=1$

- $A3$
- $P7: (1+1)+1=3$

- $s_0$
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- Build the graph iteratively keeping the minimum cost when a literal $p$ re-appears
  - (similar to planning graphs, stop when no changes arise)

Additive heuristic $h_{\text{add}}$: sum the cost of preconditions +1

- $P1: 0$
- $P2: 0$
- $P3: 0$
- $P4: 0$
- $P5: (0+0)+1=1$
- $P6: (0+0)+1=1$
- $P7: (1+1)+1=3$

$s_0$
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- Build the graph iteratively keeping the minimum cost when a literal $p$ re-appears

Additive heuristic $h_{\text{add}}$: sum the cost of preconditions +1

Max heuristic $h_{\text{max}}$: max cost of precond +1
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- Planning graph
  - Computing the graph has **polynomial** complexity

- Empty list of negative effects
  - Finding a solution to the relaxed planning task is **NP-complete**

- We can estimate it!
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}$

- Additive heuristic $h_{\text{add}}$: sum the cost of preconditions
- Max heuristic $h_{\text{max}}$: max cost of preconditions

Observation 1: These heuristics assume goal independence, therefore miss useful information
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- Note: literals appear at most once in this graph; the iteration in which they appear is a lower-bound of the estimated cost.

Additive heuristic $h_{\text{add}}$: sum the cost of preconditions +1

Max heuristic $h_{\text{max}}$: max cost of precond +1
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- Additive heuristic $h_{\text{add}}$: sum the cost of preconditions
- Max heuristic $h_{\text{max}}$: max cost of preconditions

- Observation 1: These heuristics assume goal independence, therefore miss useful information
- Observation 2: Planning graphs keep track of how actions interact, and look like the graphs we examined
Planning graphs

- Note: literals are structured in increasingly larger layers which also keep track of how actions interact.
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$

- Additive heuristic $h_{\text{add}}$: sum the cost of preconditions
- Max heuristic $h_{\text{max}}$: max cost of preconditions

- **Observation 1:** These heuristics assume goal independence, therefore miss useful information

- **Observation 2:** Planning graphs keep track of how actions interact, and look like the graphs we examined

- **FF Heuristic:** Let’s apply the empty delete list relaxation to planning graphs!

  [Hoffmann, Nebel 2001]
Relaxed planning task: FF

- Assume an empty list of negative effects
Relaxed planning task: FF

- Assume an empty list of negative effects
- No negative literals
Relaxed planning task: FF

- Assume an empty list of negative effects
- No negative literals → No mutual constraints
Relaxed planning task: FF

- Extracting a solution has **polynomial** complexity: pick actions for each sub-goal in a single sweep

Note: this is actually not a very good example because we have used negative preconditions (did anybody notice? :-)

Relaxed planning task: FF

- Extracting a solution has **polynomial** complexity: pick actions for each sub-goal in a single sweep

In any case, here we would have stopped at $s_1$, where we first reach the goal
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$, FF, $h^2$

- Additive heuristic $h_{\text{add}}$: sum the cost of preconditions
- Max heuristic $h_{\text{max}}$: max cost of preconditions
- FF heuristic: exploit positive interaction
- $h^2$ heuristic: same idea like $h_{\text{max}}$ but keep track of pairs of literals

Still one of the best heuristics!
Relaxed planning task: $h_{\text{add}}$, $h_{\text{max}}$, FF, $h^2$

- **Additive heuristic** $h_{\text{add}}$: 
  sum the cost of preconditions + 1
  
- **Max heuristic** $h_{\text{max}}$: 
  max cost of preconditions + 1

- **FF heuristic**: 
  exploit positive interaction

- **$h^2$ heuristic**: 
  same idea like $h_{\text{max}}$ but keep track of pairs of literals
Relaxed planning task: $h_{\text{add}}, h_{\text{max}}, \text{FF}, h^2$

- Let’s see again the performance of the Fast-downward planner in the Sokoban planning problem we examined in Lecture 3
Using PDDL planners: Sokoban

- search/downward --search "astar(blind())" <output

Plan length: 30 step(s).
Plan cost: 30
Initial state h value: 1
Expanded 1372 state(s).
Reopened 0 state(s).
Evaluated 1435 state(s).
Evaluations: 1435
Generated 3560 state(s)
Dead ends: 0 state(s).
Expanded until last jump: 1356 state(s).
Reopened until last jump: 0 state(s).
Evaluated until last jump: 1415 state(s).
Generated until last jump: 3521 state(s).
Search space hash size: 1435
Search space hash bucket count: 1543
Search time: 0s
Total time: 0s
Peak memory: 3036 KB
Using PDDL planners: Sokoban

- search/downward --search "astar(goalcount())"

Plan length: 30 step(s).
Plan cost: 30
Initial state h value: 1
Expanded 1298 state(s).
Reopened 0 state(s).
Evaluated 1365 state(s).
Evaluations: 1365
Generated 3370 state(s)
Dead ends: 0 state(s).
Expanded until last jump: 1295 state(s).
Reopened until last jump: 0 state(s).
Evaluated until last jump: 1361 state(s).
Generated until last jump: 3365 state(s).
Search space hash size: 1365
Search space hash bucket count: 1543
Search time: 0s
Total time: 0s
Peak memory: 3040 KB
Using PDDL planners: Sokoban

- search/downward --search "astar(hmax())"

Plan length: 30 step(s).
Plan cost: 30
Initial state h value: 5
Expanded 139 state(s).
Reopened 0 state(s).
Evaluated 176 state(s).
Evaluations: 176
Generated 364 state(s)
Dead ends: 21 state(s).
Expanded until last jump: 133 state(s).
Reopened until last jump: 0 state(s).
Evaluated until last jump: 166 state(s).
Generated until last jump: 351 state(s).
Search space hash size: 176
Search space hash bucket count: 193
Search time: 0s
Total time: 0s
Peak memory: 3052 KB
Using PDDL planners: Sokoban

- search/downward --search "astar(add())" <output

```plaintext
Plan length: 30 step(s).
Plan cost: 30
Initial state h value: 9
Expanded 93 state(s).
Reopened 0 state(s).
Evaluated 142 state(s).
Evaluations: 142
Generated 253 state(s)
Dead ends: 18 state(s).
Expanded until last jump: 72 state(s).
Reopened until last jump: 0 state(s).
Evaluated until last jump: 103 state(s).
Generated until last jump: 198 state(s).
Search space hash size: 142
Search space hash bucket count: 193
Search time: 0s
Total time: 0s
Peak memory: 3052 KB
```
Using PDDL planners: Sokoban

- search/downward --search "lazy_greedy(ff())" <output

Plan length: 30 step(s).
Plan cost: 30
Initial state h value: 5
Expanded 126 state(s).
Reopened 0 state(s).
Evaluated 145 state(s).
Evaluations: 145
Generated 335 state(s)
Dead ends: 18 state(s).
Search time: 0s
Total time: 0s
Peak memory: 3052 KB
Lecture 1: Game-inspired competitions for AI research, AI decision making for non-player characters in games

Lecture 2: STRIPS planning, state-space search

Lecture 3: Planning Domain Definition Language (PDDL), using an award winning planner to solve Sokoban

Lecture 4: Planning graphs, domain independent heuristics for STRIPS planning

Lecture 5: Employing STRIPS planning in games: SimpleFPS, iThinkUnity3D, SmartWorkersRTS

Lecture 6: Planning beyond STRIPS
References