structural patterns

abstraction heuristics: simplify problem by reducing the number of states

structural pattern: simplify the problem by turning it into one that can be solved efficiently

implementation: action sequencing

---

action sequencing

```
a: -x-y-z-w ⇒ yw
b: ...
```

action `a` has two effects

split into a sequence of two actions
one for each effect

how to sequence an action

```
a: -x-y-z-w ⇒ yw
```

split into:

```
a1: -x-y ⇒ y
a2: -xy-z-w ⇒ w
```

`a` makes `yw` true from `xyzw`

do the same incrementally,
following the order `xyzw`:

- obtain `y` from `xy`
- then `w` from `xyzw`

[note] To be read as: make some variables true (or false) from the previous value of some other variables.

The reason why `a` has precondition `¬y` while `a2` has `y` is explained in the next slide.

Actions could be sequenced in other ways, but this particular one has been proved to have certain good properties.

the action and its sequence

```
a: -x-y-z-w ⇒ yw
```

split into:

```
a1: -x-y ⇒ y
a2: -xy-z-w ⇒ w
```

sequence `a1,a2` same as `a`

```
a:     -x-y-z-w        ⇒       -xy-zw
a1,a2: -x-y-z-w ⇒ -xy-z-w ⇒ -xy-zw
```

if `a` can be executed in a state
also `a1,a2` can, with same effects

[note] The aim of sequencing is to emulate `a` by `a1,a2`. This is why `a2` has precondition `y` instead of `¬y` like `a`. The intention is that `a2` is executed after `a1`, and `a1` makes `y` true.

---

other plans

```
a: -x-y-z-w ⇒ yw
```

split into:

```
a1: -x-y ⇒ y
a2: -xy-z-w ⇒ w
```

instead of `a`, execute `a1,a2`

`a1` can also be executed when `a` cannot
same for `a2`

is this a problem?

[note] The action is equivalent to the sequence that replaces it. However, the single actions of the sequence can also be executed independently.

admissibility

```
a: -x-y-z-w ⇒ yw
```

split into:

```
a1: -x-y ⇒ y
a2: -xy-z-w ⇒ w
```

`a` is the same as `a1,a2`

plans for the original instance are still plans

admissible heuristics, if:
cost of `a` = cost of `a1+a2`

example: each action gets half the cost

[note] If `a` can be applied, then also `a1,a2` can, and the result is the same. The action `a1` can also be applied alone in some states where `a` is inapplicable, and `a2` can be applied even if `a1` was not. This is not a problem. What is important is that the plans for the original problem are mapped into plans after sequencing. The aim is not to preserve the domain exactly, but to simplify the problem in order to obtain an estimate of the cost.

---

admissible! done?

just being admissible is not enough

- admissible? yes
- easy to calculate?
  randomly sequencing actions: no

further elaboration needed

[note] Sequencing actions makes for an admissible heuristics, but not one polynomial to calculate. Something more is necessary.

further requirements

1. sequence all actions according to the same order
2. allow the order to be partial, even non-transitive
3. remove variables incomparable with all others
4. use some specific orders
5. abstract values (collect multiple values in one)

[note] This is not an algorithm. These are not Steps 1-5 of an algorithm. Points 1-5 are requirements on the sequencing of actions that make the resulting heuristics polynomial-time.

Now, these points are briefly described. Before: sequencing, in general.

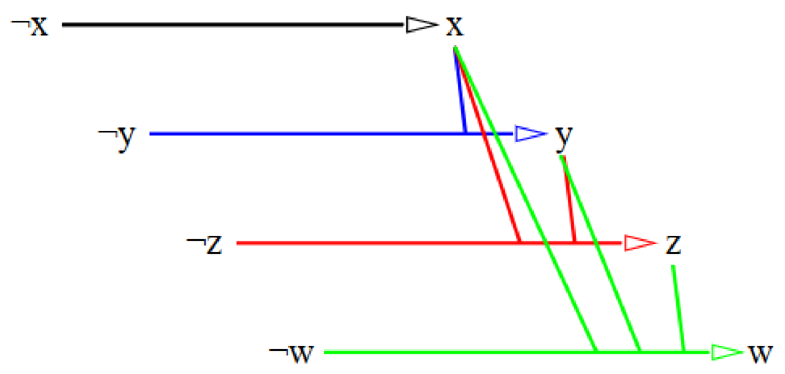
sequencing, in general

example: `-x-y-z-w ⇒ xyzw`
ordering `xyzw`

effects are obtained one at time, following the order

preconditions for an effect: the preceding variables

$\neg x \quad \neg y \quad \neg z \quad \neg w \longrightarrow x \quad y \quad z \quad w$



[click on image to start animation]

in general: same scheme
remove variables not in the original action
remove actions with no effect

[note] For example, if the original action does not have the precondition `¬y`, then `¬y` is removed from the diagram. The same for the effects, but an action with no effect can be removed altogether.

Now the points 1-5 for obtaining a good heuristics are described.

point 1: same variable order

```
a: -x-y-z-w ⇒ yw

a1: -x-y ⇒ y
a2: -xy-z-w ⇒ w
```

action `a` sequenced with order `xyzw`

same order `xyzw`
for another action `b`:

```
b: yz ⇒ x-z-w

b1: ⇒ x
b2: xyz ⇒ -z
b3: xy-z ⇒ -w
```

[note] Using a different ordering for sequencing each action is possible and would still result in an admissible heuristics. But using the same ordering for all allows, together with other conditions, to obtain a polynomial-time heuristics.

point 2: partial order

allow the order to be partial
even non-transitive

example: `x≤z` and `y≤z`
`x` and `y` incomparable

```
c: xyz ⇒ -x-y-z

c1: x ⇒ -x
c2: y ⇒ -y
c3: -x-yz ⇒ -z
```

effects:

- change `x` before `z`
- change `y` before `z`
- no order between changing `x` and `y`

preconditions:

- `x` is precondition to `z`
- `y` is precondition to `z`
- but `x` is not a precondition to `y` nor vice versa

[note] The order still constrains the order in which the effects are obtained, and which variables are precondition to which effect.

Since the order is partially specified, the effects can be obtained in more than one way. In this case, `c` is emulated by `c1c2c3` but also by `c2c1c3`, since both sequences obey `x≤z` and `y≤z`.

In the same way, the order tells which variables are precondition to which effects. The action generating `z` has preconditions `x` and `y` because of `x≤z` and `y≤z`. However, since `x` and `y` are not ordered, the action generating `y` does not have `x` as a precondition and vice versa.

point 3: remove incomparable variables

remove variables like in pattern database

here: remove all variables incomparable with all others

---

point 4: which orders

fix a variable `x`

the order is `x≤y`, `x≤z`, `x≤w`, …
if there are actions:

```
...x...⇒...y...   or   ...⇒...x...y...
...x...⇒...z...   or   ...⇒...x...z...
...x...⇒...w...   or   ...⇒...x...w...
```

etc.

all other variables are removed

sequence *all* actions with this order
result: an admissible heuristics

[note] This restriction can be defined in terms of the forks in the causal graph of the problem, but such concepts are not in this course.

point 5: restrict the domain

refine the previous point

the order is $x \le y$, $x \le z$, $x \le w$, …
if there are actions:

  …x…⇒…y… or …⇒…x…y…
  …x…⇒…z… or …⇒…x…z…
  …x…⇒…w… or …⇒…x…w…

etc.

furthermore: x has only two possible values

make multiple values become one
example: 1,2,3 become 0; the other values 4,5,6,7,8,9 become 1

[note] The previous examples have only binary variables (true/false), so the possible values are two by definition. In general, a variable may take an arbitrary number of values.

If the possible values for a variable x are not two, they need to be reduced. This is done by replacing sets of values with a single one, similarly to how states are abstracted.

wrap up

for each variable x,
a partial order is defined
sequence actions with this order
make x have only two possible values
optimal plan can be calculated in polynomial time

so: an heuristics for each variable

distributing the cost of a among these heuristics:
these heuristics are additive

[note] The heuristics are additive only distributing the cost of each action among them. Otherwise, they are only admissible. They are polynomial-time regardless.

inverted forks

another order for x is: y≤x z≤x,…
for actions …y…⇒…x…, or …⇒…y…x…, etc.
constant number of possible values for x

still polynomial-time

---

abstraction

remove variable = grouping of states

actions not preserved (unlike pattern database, merge&shrink)
cost of reachability preserved
cost of going from s to s' can only decrease