Lab 4 matge373

4.1

Lord of the Rings:

The starting state (S0) is the following:

undelivered(one-ring),
destination(one-ring, mount-doom),
delivery-speed(one-ring, slow),
at(hobbit, bag-end),
holding(hobbit, one-ring)

and the task is: <make-delivery(one-ring)>

Here, only **undelivered(...)**, **at(...)** and **holding(...)** will change over the different states, so I will only include those in my description of the problem.

S0	**make-delivery(one-ring)**

S0 is: undelivered(one-ring)
at(hobbit, bag-end),
holding(hobbit, one-ring)

S0	**walk(carrier, from, to)**	**deliver(pkg, carrier, to)**

S0 is: undelivered(one-ring)
at(hobbit, bag-end),
holding(hobbit, one-ring)

walk(hobbit, bag-end, mount-doom)	**S1**	**deliver(pkg, carrier, to)**

S1 is: undelivered(one-ring)
at(hobbit, mount-doom),
holding(hobbit, one-ring)

walk(hobbit, bag-end, mount-doom)	**deliver(one-ring, hobbit, mount-doom)**	**S2**

S2 is: ¬undelivered(one-ring)
at(hobbit, mount-doom),
¬holding(hobbit, one-ring)

The ring has now been delivered!

<u>Star Wars:</u>

The starting state (S0) is the following:

undelivered(death-star-plans),
destination(death-star-plans, hidden-rebel-base),
delivery-speed(death-star-plans, fast),
at(farm-boy, desert-far-away),
holding(farm-boy,death-star-plans),
at(old-ship, desert-far-away)

and the task is: <make-delivery(death-star-plans)>

Here, the atoms that star the same over all states are: **destination(...)** and **delivery-speed(...)**, which therefore will not be included in the different state changes below.

S0	**make-delivery(death-star-plans)**

S0 is: undelivered(death-star-plans)
at(farm-boy, desert-far-away),
holding(farm-boy, death-star-plans),
at(old-ship, desert-far-away)

S0	**enter-vehicle(carrier, vehicle, from)**	**travel(vehicle, from, to)**	**leave-and-deliver(carrier, pkg, vehicle, to)**

S0 is: undelivered(death-star-plans)
at(farm-boy, desert-far-away),
holding(farm-boy, death-star-plans),
at(old-ship, desert-far-away)

enter-vehicle(farm-boy, vehicle, from)	**S1**	**travel(vehicle, from, to)**	**leave-and-deliver(carrier, pkg, vehicle, to)**

S1 is: undelivered(death-star-plans)
¬at(farm-boy, desert-far-away),
in(farm-boy, old-ship),
holding(farm-boy, death-star-plans),
at(old-ship, desert-far-away)

enter-vehicle(farm-boy, vehicle, from)	**travel(old-ship, desert-far-away, hidden-rebel-base)**	**S2**	**leave-and-deliver(carrier, pkg, vehicle, to)**

S2 is: undelivered(death-star-plans)
in(farm-boy, old-ship),
holding(farm-boy, death-star-plans),
¬at(old-ship, desert-far-away),
at(old-ship, hidden-rebel-base)

enter-vehicle(farm-boy, vehicle, from)	**travel(old-ship, desert-far-away, hidden-rebel-base)**	**S2**	**leave-vehicle(carrier, vehicle, loc)**	**deliver(pkg, carrier, to)**

S2 is the same as above, leave-and-deliver was simply decomposed.

enter-vehicle(farm-boy, vehicle, from)	**travel(old-ship, desert-far-away, hidden-rebel-base)**	**leave-vehicle(farm-boy, old-ship, hidden-rebel-base)**	**S3**	**deliver(pkg, carrier, to)**

S3 is: undelivered(death-star-plans)
¬in(farm-boy, old-ship),
at(farm-boy, hidden-rebel-base),
holding(farm-boy, death-star-plans),
at(old-ship, hidden-rebel-base)

enter-vehicle(farm-boy, vehicle, from)	**travel(old-ship, desert-far-away, hidden-rebel-base)**	**leave-vehicle(farm-boy, old-ship, hidden-rebel-base)**	**deliver(death-star-plans, farm-boy, hidden-rebel-base)**	**S4**

S4 is: ¬undelivered(death-star-plans)
at(farm-boy, hidden-rebel-base),
¬holding(farm-boy, death-star-plans),
at(old-ship, hidden-rebel-base)

The death star plans have now been delivered!

Questions

1. Compare building the HTN domain to what you did previously in PDDL. Which parts were easier, if any? Which parts were harder? Why?

To me, the HTN was quite intuitive once you got the hang of it. It is simple in the regard of being divided into methods and operators, which have clear descriptions of what they do. It is hard to say if there were any parts that were easier, since I had the old PDDL domain as a “cheat sheet” of what needed to be translated to the HTN. But generally, I find it rather easy in the way that operators and methods are defined. Creating an operator that does something by precondition→deletelist→addlist is quite simple, and as long as you know what needs to be done there is no problem doing it. The method part is also very nice and intuitive.

The part that I found harder was the troubleshooting, which probably does not come as a surprise to anyone. Since there is not really anything there to help you if you write something wrong or if a specific line would lead to an infinite recursion, it can be quite tricky finding the bug. I did not have any major issues with this, but it was still annoying when these errors occurred.

2. How does the run time of JSHOP2 scale with larger problems?

It scales very well with larger problems. I have created a “medium” size problem which would finish within less than a second, and looking at the GUI it reached around 4600 actions in that time. Running a problem twice as large still finishes in less than a second, but the GUI does not show the action plan due to stack overflow (probably because the problem and resulting action plan is too large?). Given this, my JSHOP domain should have no issue whatsoever to reach 20 000 actions in a minute, while still finishing it very fast.

3. Is JSHOP2 faster/slower than the competition planners from previous labs, given your own particular PDDL and HTN formalizations of the domain? How much?

JSHOP2, according to this lab, is faster than the competition planners from the previous labs, by quite a lot it seems. It is difficult to visualize since the problems are not the exact same, and I don’t have a problem generator that creates and compares similar problems. Since the JSHOP planner seems to be completely unphased with creating plans for problems with over 10 000 actions, I think it is safe to say that it solves problems very very fast.