CS 520: Planning Example for Wolf/Goat/Cabbage

16:198:520
Instructor: Wes Cowan

Consider the classic problem: you are on one side of a river with a wolf, a goat, and a cabbage. You want to transport all three to the other side of the river, but you can only transport one object at a time. You cannot leave the wolf and the goat alone, or the cabbage and the goat alone; you are the only thing keeping them from eating each other. How can you transport everything from one side of the river to the other? How can we get a *computer* to solve the problem? One thing we can do is to try to formulate it in terms of a **Planning Domain Definition Language (PDDL)**.

1 Formulation

We may define a set of objects relevant to the problem, in particular: **Wolf**, **Goat**, **Cabbage**, **You**. It may additionally be convenient to define the locations **Bank1** and **Bank2**. Once we have these, we may define the *state* of the problem in terms of relationships between these objects. In particular, we have an initial state:

$$\text{Initial State: } \text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)}.$$

We may specify the *goal* state similarly:

$$\text{Goal State: } \text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)}.$$

The state of the problem at any point in time may be framed in terms of a conjunction of these kinds of relationships; the state as a logical expression expands or contracts depending on the actions taken. It is worth noting that in general, the goal state does not need to be a complete specification of a state. For instance, if we only wanted to transport the goat and the cabbage to the other side (for instance if the wolf is not our wolf, just some random wolf prowling the bank), we need only specify $\text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)}$ - any state that satisfies this expression, *regardless of where the wolf is!* is treated as a successful goal state.

We may then define a set of actions on these state descriptions, modeling actions that can be taken in the problem. In particular, we are interested in **Transport**, which allows us to take an object from one side to the other: **Transport(Object, Loc1, Loc2)**. Note that there are some states in which a given move action may not be applicable - the current state must satisfy the *preconditions* of the action, in order to execute it. In particular, in order to transport an Object from Loc1 to Loc2, not only must Object be in Loc1, but *you* must be in Loc1 as well.

$$\text{Preconditions of } \textbf{Transport}\text{: } \text{On(Obj, Loc1)} \wedge \text{On(You, Loc1)}.$$

What is the *effect* of this action? On(Obj, Loc1) and On(You, Loc1) are no longer true, and $\text{On(Obj, Loc2)} \wedge \text{On(You, Loc2)}$ become true,

$$\text{Effect of } \textbf{Transport}\text{: } \neg\, \text{On(Obj, Loc1)} \wedge \neg\, \text{On(You, Loc1)} \wedge \text{On(Obj, Loc2)} \wedge \text{On(You, Loc2)}.$$

New statements that are true can essentially be thought of as being tacked on to the current state description. Statements that are now false may be thought as being *deleted* from the state description. Hence, calling Transport(Goat, Bank1, Bank2) will result in a new state:

$$\text{On(Wolf, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank2)} \wedge \text{On(Goat, Bank2)}.$$

At this point, we might introduce some extra bookkeeping to introduce efficiency. For instance, calling Transport(Obj, Loc1, Loc1) may be a perfectly executable statement - but it has no serious effect or advantage to consider, and in fact could potentially make searching the whole state space that much less efficient. We could avoid this by adding in the precondition that Loc1 $\neq$ Loc2.

Similarly, it is worth noting that we could affect the operation of going back to the other side simply by calling Transport on You (i.e., the object you transport is yourself). However, we can gain some clarity simply by introducing a new operation **Move(Loc1, Loc2)**:

$$\text{Move(Loc1, Loc2)}$$
$$\text{Preconditions for Move: On(You, Loc1)} \wedge \text{Loc1} \neq \text{Loc2}$$
$$\text{Effect of Move: } \neg \text{ On(You, Loc1)} \wedge \text{On(You, Loc2)}.$$

At this point, we have enough structure to begin affecting a search: we can start at the initial state, and sequentially explore possible actions to take from there and the resulting states, either in a BFS or DFS type search (or, as we will discuss, A^*).

However, one thing has not yet been addressed - the restriction that the wolf and goat cannot be alone together, and that the goat and the cabbage cannot be alone together. This can be incorporated into the problem in one of two ways: one, we could extend the preconditions on Transport and Move to account for this (*how*?); alternately, we could introduce the notion of states to avoid, for instance any state that satisfies:

$$\text{On(Goat, Loc1)} \wedge \text{On(Wolf, Loc1)} \wedge \text{On(You, Loc2)} \wedge \text{Loc1} \neq \text{Loc2}$$
$$\text{or}$$
$$\text{On(Goat, Loc1)} \wedge \text{On(Cabbage, Loc1)} \wedge \text{On(You, Loc2)} \wedge \text{Loc1} \neq \text{Loc2}$$

Alternately, these could be combined into a single logical description of a state to avoid. Note that since Loc1, Loc2 aren't necessarily specified explicitly (though they could be, expanding the bad-state definition), there may be some work to determine when the preconditions or terminal states are *satisfied* as well.

At this point, we have defined the problem to the point of being able to search, to try to find paths from the initial state to the goal state. Approaching this in terms of BFS, the reachable states from the initial state are

$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank2)}$$
$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank2)}$$
$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)}$$
$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank2)}$$

But of these, three satisfy the 'states to avoid' condition, leaving only:

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank2)}.$$

At this point, the applicable actions are either move or transport. Avoiding loops in our search space leads to

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)}.$$

This search process can be continued, ultimately arriving at a total path of:

Transport(Goat, Bank1, Bank2)
Move(Bank2, Bank1)
Transport(Wolf, Bank1, Bank2)
Transport(Goat, Bank2, Bank1)
Transport(Cabbage, Bank1, Bank2)
Move(Bank2, Bank1)
Transport(Goat, Bank1, Bank2).

This is not necessarily the only path, but it is correct and discoverable via search.

2 Backwards Search

In some cases, it may be useful to consider the problem of searching backwards - given that we want to end up at the goal, how could we have arrived there? Note in particular that in this case, we can consider inverting actions, and construct what the state must've been prior to actions being taken.

If our goal state is

$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)},$$

then our immediately previous state must've been one of the following:

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank1)}$$
$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank1)}$$
$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)}$$
$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank2)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank1)},$$

but of these only one is acceptable as a state of the problem:

$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank1)}.$$

If we consider what the state must've been prior to this, we have a couple of options, but avoiding loops (i.e., carrying the goat back), we have

$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)}.$$

Prior to this, (again avoiding loops), we must've had one of the following:

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank1)}$$
$$\text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)},$$

and we could again consider completing this process to reconstruct a full path.

One important aspect of backwards search - it may not immediately be apparently which object was used in a given action. For instance, if we are talking planes transporting cargo, and the goal state is the cargo in a given location, then a plane must've delivered it there, but we have no way of being sure which plane. One way to approach this would be to consider all possible planes and work backwards from that, another might be to introduce a 'dummy' variable plane, and execute backwards operations on that, until it becomes clear which plane the dummy variable must have been. Again, satisfiability orbits at the periphery.

3 Heuristics

As is ever the case with search algorithms, the worst possible case yields an exponential runtime in terms of the length of the path to the goal state. We could try to improve upon this by using an informed search algorithm like A^*, but what would we use for a heuristic? There are possible ways of measuring the distance between state descriptions (comparing the number of satisfied clauses, etc) but two frequent approaches are to consider *relaxations* of the original problem, problems that are easier to solve and therefore yield a lower bound on the number of steps needed to solve the original problem. Both of these consider the 'abstract' formalized state-search problem, rather than the modeled problem itself. This can lead to some confusion in interpretation, but the results are still meaningful.

3.1 Ignoring Deletions

Consider the effect of actions if we ignore negated clauses. In this case, actions will only ever *increase* the size of the state description, taking on ever more (potentially contradictory) clauses. This means that we will reach a state that satisfied the goal sooner than we would be able to otherwise. As applied to this problem, we might apply **Transport(Cabbage, Bank1, Bank2)** to the initial state, yielding a state of

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)}.$$

Obviously this state description describes something that cannot occur, but is still *functional* in terms of our formalized representation and operation. Note then that at this point, with this state, we are still able to call Transport(Goat, Bank1, Bank2) and Transport(Wolf, Bank1, Bank2), yielding a final state of (ignoring redundancies)

$$\text{On(Wolf, Bank1)} \wedge \text{On(Goat, Bank1)} \wedge \text{On(Cabbage, Bank1)} \wedge \text{On(You, Bank1)} \wedge \text{On(Cabbage, Bank2)} \wedge \text{On(You, Bank2)} \wedge \text{On(Wolf, Bank2)} \wedge \text{On(Goat, Bank2)}.$$

At this point, note that we have reached a *(formal, abstract)* state that satisfies the goal state! And we have done so in three steps. What we can conclude from this is that since the original problem can be no harder than this relaxed problem (since ignoring deletions means the goal state will be satisfied **sooner**), the original problem must take *at least three steps* to get from the start to the end.

Using the solution to the relaxed problem (which may be solved by BFS, IDFS, etc) does not yield valid results for the original problem (generally), but the length of the path / number of steps in the resulting schedule can be informative, and be used as an admissible heuristic for A^* in the original problem.

3.2 Ignoring Preconditions / Bad States

Consider the effect of ignoring preconditions. Preconditions potentially restrict what actions are applicable at any given time. By ignoring the preconditions, we are essentially expanding the set of available actions. With an increased set of actions, we must be able to reach something satisfying the goal state at least as quickly as we could paying attention to the preconditions. Applying this line of thought in this case we can consider calling Transport(Goat, Bank1, Bank2), and then (ignoring preconditions and potential bad states) Transport(Wolf, Bank1, Bank2), Transport(Cabbage, Bank1, Bank2). These are not 'allowable' moves in the original problem, but in this relaxed version of the problem they are completely acceptable, arriving at a state that satisfies the goal.

Again, we may conclude that the original problem may be solved in no fewer than three steps.