Search: A* relaxations

How do we get good heuristics? Just relax...

Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.

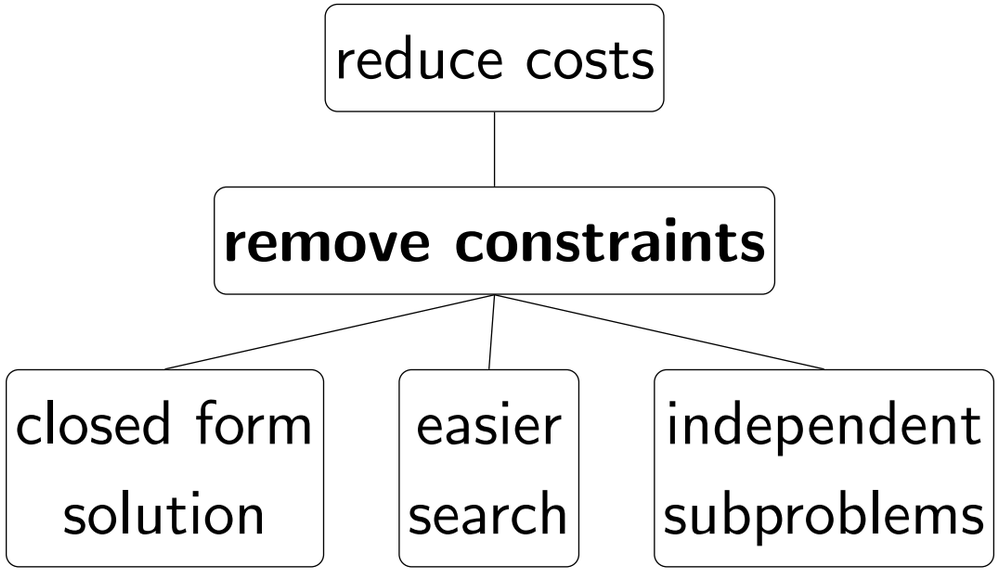
Key idea: relaxation

Constraints make life hard. Get rid of them.

But this is just for the heuristic!

- So far, given a heuristic $h(s)$, we can run A* using it and get a savings which depends on how large $h(s)$ is. However, we've only seen two heuristics: $h(s) = 0$ and $h(s) = \text{FutureCost}(s)$. The first does nothing (gives you back UCS), and the second is hard to compute.
- What we'd like to do is to come up with a general principle for coming up with heuristics. The idea is that of a **relaxation**: instead of computing $\text{FutureCost}(s)$ on the original problem, let us compute $\text{FutureCost}(s)$ on an easier problem, where the notion of easy will be made more formal shortly.
- Note that coming up with good heuristics is about **modeling**, not algorithms. We have to think carefully about our problem domain and see what kind of structure we can exploit in it.

Relaxation overview

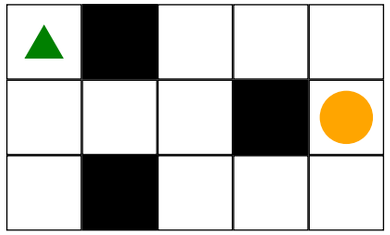


combine heuristics using max

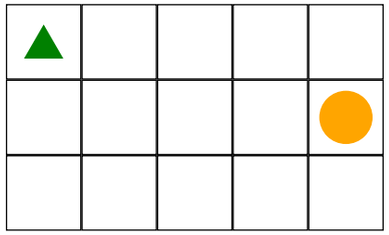
Closed form solution

Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

e.g., $h((1, 1)) = 5$

- Here's a simple example. Suppose states are positions (r, c) on the grid. Possible actions are moving up, down, left, or right, provided they don't move you into a wall or off the grid; and all edge costs are 1. The start state is at the triangle at $(1, 1)$, and the end state is the circle at position $(2, 5)$.
- With an arbitrary configuration of walls, we can't compute FutureCost(s) except by doing search. However, if we just **relaxed** the original problem by removing the walls, then we can compute FutureCost(s) in **closed form**: it's just the Manhattan distance between s and s_{end}. Specifically, ManhattanDistance$((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$.

Easier search

Example: original problem

Start state: 1

Walk action: from s to $s+1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

Constraint: can't have more tram actions than walk actions.

State: (location, **#walk - #tram**)

Number of states goes from $O(n)$ to $O(n^2)$!

- Let's revisit our magic tram example. Suppose now that a decree comes from above that says you can't have take the tram more times than you walk. This makes our lives considerably worse, since if we wanted to respect this constraint, we have to keep track of additional information (augment the state).
- In particular, we need to keep track of the number of walk actions that we've taken so far minus the number of tram actions we've taken so far, and enforce that this number does not go negative. Now the number of states we have is much larger and thus, search becomes a lot slower.

Easier search

Example: relaxed problem

Start state: 1

Walk action: from s to $s+1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

~~Constraint: can't have more tram actions than walk actions.~~

Original state: (location, **#walk - #tram**)

Relaxed state: location

- What if we just ignore that constraint and solve the original problem? That would be much easier/faster. But how do we construct a consistent heuristic from the solution from the relaxed problem?

Easier search

- Compute relaxed $\text{FutureCost}_{\text{rel}}(\text{location})$ for **each** location $(1, \ldots, n)$ using dynamic programming or UCS

Example: reversed relaxed problem

Start state: n

Walk action: from s to $s-1$ (cost: 1)

Tram action: from s to $s/2$ (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem (equivalent to future costs in relaxed problem!)

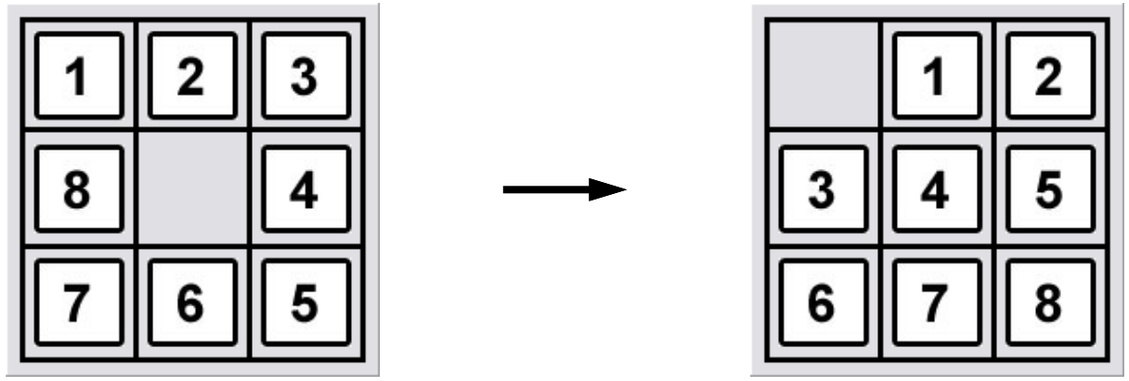
- Define heuristic for original problem:

$$h((\text{location, \#walk-\#tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

- We want to now construct a heuristic $h(s)$ based on the future costs under the relaxed problem.
- For this, we need the future costs for all the relaxed states. One straightforward way to do this is by using dynamic programming. However, if we have cycles, then we need to use uniform cost search.
- But recall that UCS only computes the past costs of all states up until the end. So we need to make two changes. First, we simply don't stop at the end, but keep on going until we've explored all the states. Second, we define a **reversed relaxed problem** (where all the edges are just reversed), and call UCS on that. UCS will return past costs in the reversed relaxed problem which correspond exactly to future costs in the relaxed problem.
- Finally, we need to construct the actual heuristic. We have to be a bit careful because the state spaces of the relaxed and original problems are different. For this, we set the heuristic $h(s)$ to the future cost of the relaxed version of s.
- Note that the minimum cost returned by A* (UCS on the modified problem) is the true minimum cost minus the value of the heuristic at the start state.

Independent subproblems

[8 puzzle]



Original problem: tiles cannot overlap (constraint)

Relaxed problem: tiles can overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form

Key idea: independence

Relax original problem into independent subproblems.

- So far, we've seen that some heuristics $h(s)$ can be computed in closed form and others can be computed by doing a cheaper search. But there's another way to define heuristics $h(s)$ which are efficient to compute.
- In the 8-puzzle, the goal is to slide the tiles around to produce the desired configuration, but with the constraint that no two tiles can occupy the same position. However, we can throw that constraint out the window to get a relaxed problem. Now, the new problem is really easy, because the tiles can now move **independently**. So we've taken one giant problem and turned it into 8 smaller problems. Each of the smaller problems can now be solved separately (in this case, in closed form, but in other cases, we can also just do search).
- It's worth remembering that all of these relaxed problems are simply used to get the value of the heuristic $h(s)$ to guide the full search. The actual solutions to these relaxed problems are not used.

General framework

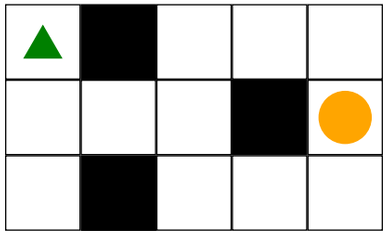
Removing constraints

(knock down walls, walk/tram freely, overlap pieces)

Reducing edge costs

(from ∞ to some finite cost)

Example:

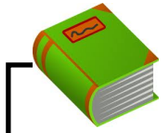


Original: $\text{Cost}((1,1),\text{East}) = \infty$

Relaxed: $\text{Cost}_{\text{rel}}((1,1),\text{East}) = 1$

- We have seen three instances where removing constraints yields simpler solutions, either via closed form, easier search, or independent subproblems. But we haven't formally proved that the heuristics you get are consistent!
- Now we will analyze all three cases in a unified framework. Removing constraints can be thought of as adding edges (you can go between pairs of states that you weren't able to before). Adding edges is equivalent to reducing the edge cost from infinity to something finite (the resulting edge cost).

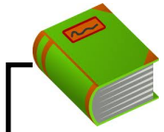
General framework



Definition: relaxed search problem

A **relaxation** P_{rel} of a search problem P has costs that satisfy:

$$\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a).$$

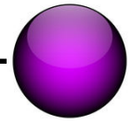


Definition: relaxed heuristic

Given a relaxed search problem P_{rel}, define the **relaxed heuristic** $h(s) = \text{FutureCost}_{\text{rel}}(s)$, the minimum cost from s to an end state using $\text{Cost}_{\text{rel}}(s, a)$.

- More formally, we define a relaxed search problem as one where the relaxed edge costs are no larger than the original edge costs.
- The relaxed heuristic is simply the future cost of the relaxed search problem, which by design should be efficiently computable.

General framework



Theorem: consistency of relaxed heuristics

Suppose $h(s) = \text{FutureCost}_{\text{rel}}(s)$ for some relaxed problem P_{rel}.

Then $h(s)$ is a consistent heuristic.

Proof:

$$h(s) \leq \text{Cost}_{\text{rel}}(s, a) + h(\text{Succ}(s, a)) \text{ [triangle inequality]}$$
$$\leq \text{Cost}(s, a) + h(\text{Succ}(s, a)) \text{ [relaxation]}$$

- We now need to check that our defined heuristic $h(s)$ is actually consistent, so that using it actually will yield the minimum cost path of our original problem (not the relaxed problem, which is just a means towards an end).
- Checking consistency is actually quite easy. The first inequality follows because $h(s) = \text{FutureCost}_{\text{rel}}(s)$, and all future costs correspond to the minimum cost paths. So taking action a from state s better be no better than taking the best action from state s (this is all in the search problem P_{rel}).
- The second inequality follows just by the definition of a relaxed search problem.
- The significance of this theorem is that we only need to think about coming up with relaxations rather than worrying directly about checking consistency.

Tradeoff

Efficiency:

$h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute

Closed form, easier search, independent subproblems

Tightness:

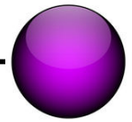
heuristic $h(s)$ should be close to FutureCost(s)

Don't remove too many constraints

- How should one go about designing a heuristic?
- First, the heuristic should be easy to compute. As the main point of A* is to make things more efficient, if the heuristic is as hard as to compute as the original search problem, we've gained nothing (an extreme case is no relaxation at all, in which case $h(s) = \text{FutureCost}(s)$).
- Second, the heuristic should tell us some information about where the goal is. In the extreme case, we relax all the way and we have $h(s) = 0$, which corresponds to running UCS. (Perhaps it is reassuring that we never perform worse than UCS.)
- So the art of designing heuristics is to balance informativeness with computational efficiency.

Max of two heuristics

How do we combine two heuristics?



Proposition: max heuristic

Suppose $h_1(s)$ and $h_2(s)$ are consistent.

Then $h(s) = \max\{h_1(s), h_2(s)\}$ is consistent.

Proof: exercise

- In many situations, you'll be able to come up with two heuristics which are both reasonable, but no one dominates the other. Which one should you choose? Fortunately, you don't have to choose because you can use both of them!
- The key point is the max of two consistent heuristics is consistent. Why max? Remember that we want heuristic values to be larger. And furthermore, we can prove that taking the max leads to a consistent heuristic.
- Stepping back a bit, there is a deeper takeaway with A* and relaxation here. The idea is that when you are confronted with a difficult problem, it is wise to start by solving easier versions of the problem (being an optimist). The result of solving these easier problems can then be a useful guide in solving the original problem.
- For example, if the relaxed problem turns out to have no solution, then you don't even have to bother solving the original problem, because a solution can't possibly exist (you've been optimistic by using the relaxation).