CS3243 **AY20/21 Sem1**

Lecture 4: September 2

Lecturer: MEEL, Kuldeep S. *Scribes: Song Qifeng*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

4.1 Recap

There are 2 properties of heuristics function:

- Admissibility
- Consistency

An admissible heuristics function always returns the lower bound of path cost, i.e.

$$\forall n \in graph\ G,\ h(n) \leq OPT(n).$$

A consistent heuristics function, on the other hand, is a subset of admissible heuristics function, because as shown in the previous lecture, $consistency \rightarrow admissibility$.

Also discussed in the previous lecture, we have only proved *consistency* is a sufficient condition for an A* graph search being optimal, yet have not proved its necessity.

$$consistency \rightarrow optimal\ A^*\ graph\ search.$$

We will prove that *admissibility* is a sufficient condition for an A* tree search being optimal in the next tutorial.

$$admissibility \rightarrow optimal\ A^*\ tree\ search.$$

4.2 How to design admissible heuristic functions

4.2.1 Generate admissible heuristics from relaxed problems

Here we discuss one of the most popular approaches to generate admissible heuristics functions - generating admissible heuristics from relaxed problems.

Let's take the 8-puzzle as an example to show how this is done.

The 8-puzzle rule can be put into formal language as following:

A tile can be moved from square A to square B iff:

A is horizontally or vertically adjacent to B $\wedge$ *B is unoccupied*

Hence we can generate three new rules by relaxing the rule differently:

- (a) A tile can be moved from square A to B if B is unoccupied.
- (b) A tile can be moved from square A to B if A is horizontally or vertically adjacent to B.
- (c) A tile can be moved from square A to B without condition.

Rule (a), (b), and (c) creates three new relaxed problems (a), (b), and (c), each is less stricter than the original one.

In problem (b), it is obvious that $OPT_b = \Sigma\ Manhattan\ distance\ of\ a\ tile\ from\ its\ goal\ position$, and we let the first heuristic function $h_b = OPT_b$.

In problem (c), it is obvious that $OPT_c = the\ number\ of\ misplaced\ tiles$, and we let the second heuristic function $h_c = OPT_c$.

In general, idea of generating heuristic function from relaxed problems is:

- First: Put the restrictions of the problem into formal language.
- Second: Generate new problems by relaxing the restrictions in different ways.
- Third: For each relaxed problem i, let $h_i = OPT_i$.

4.2.2 Proof that heuristic functions generated from relaxed problems are both admissible and consistent

4.2.2.1 Admissibility

It is trivial that the state-space of a relaxed problem is a *supergraph* of the state-space graph of the original problem.

$\forall edge\ e,\ vertex\ v \in graph_{origin},\ e,\ v \in graph_{relaxed}$

$\rightarrow\ \forall path\ p \in graph_{origin},\ p \in graph_{relaxed}$

$\rightarrow\ OPT_{origin} \in graph_{relaxed}$

$\rightarrow\ OPT_{origin} \geq OPT_{relaxed} = h_{relaxed}$

$\rightarrow\ h_{relaxed}\ is\ admissible.$

Hence $h_{relaxed}$ is admissible.

4.2.2.2 Consistency

Let n' be a successor of n in the relaxed problem's state-space graph. Because the relaxed graph only adds edges to the original graph, therefore n' is also a successor of n in the original graph.

It is self-evident that $OPT[n] \leq \ OTP[n'] + c(n, n')$,

note that $h_{relaxed} = OPT_{relexed}$

$\rightarrow \ h_{relaxed}[n] \ \leq \ h_{relaxed}[n'] + c(n, n')$

since n *and* n' *are selected arbitrarily*

$\rightarrow \ \forall \ i \in \ graph, \ \forall \ i' \ that \ is \ a \ successor \ of \ i, \ h_{relaxed}[i] \ \leq \ h_{relaxed}[i'] + c(i, i')$

Hence $h_{relaxed}$ is consistent.

4.2.3 Trade-off between computablility and informativeness

For h_b and h_c discussed above, it can be shown that for any node n, $h_b \geq h_c$, thus we say h_b **dominates** h_c. The closer h is to OPT, the more **informative** h is.

It it obvious that OPT itself is the most informative heuristic function, however, it is not as computable as h_b and h_c, which are less informative; on the other hand, letting $h = 0$ is the most computable heuristic function, yet it is just the same as UCS, which is the least informative.

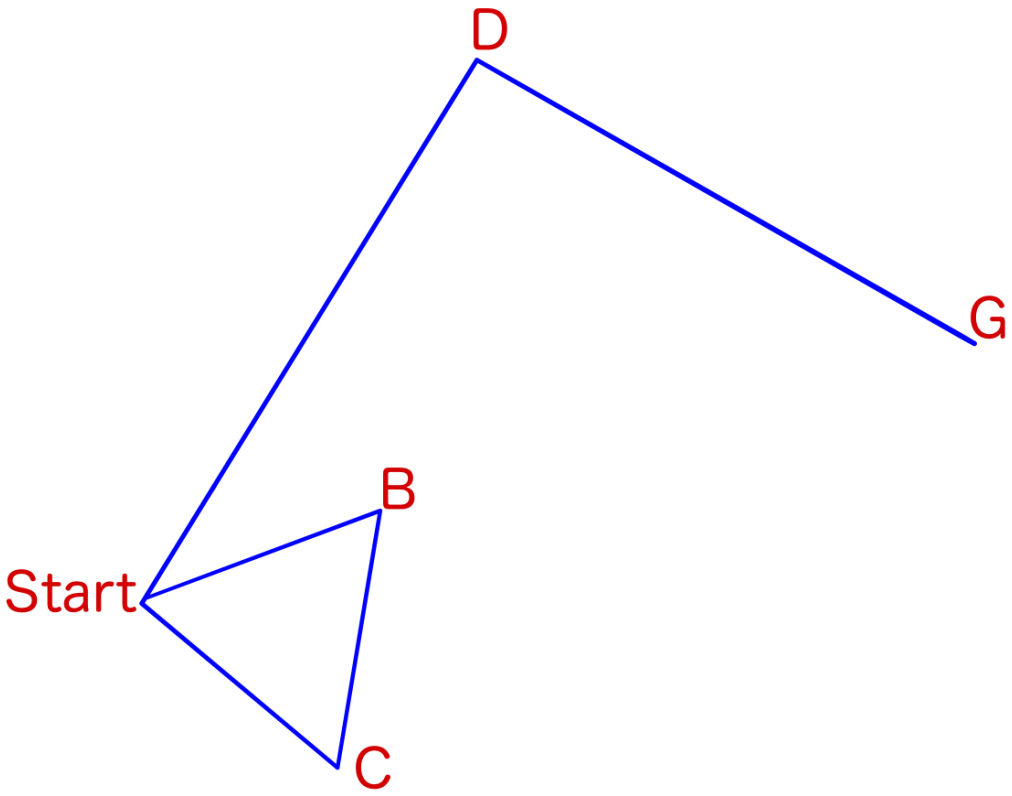
Therefore, the computability and informativeness of a heuristic function are always a trade-off.

4.3 Greedy best first search

Greedy best first search(GBFS) uses only $h(n)$ to sort the frontier, it does not care about the path cost traveled so far.

4.3.1 Completeness

GBFS is incomplete.



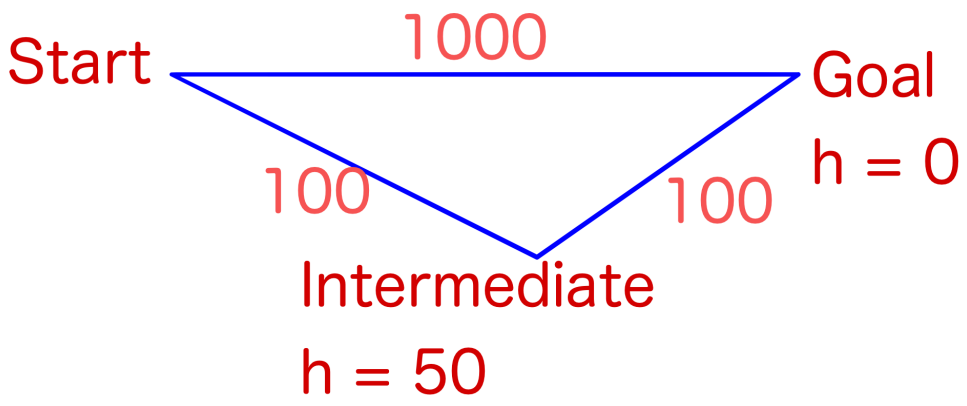
As shown in the figure above, GBFS will end up in the loop $Start - B - C - Start - \ldots$ when there exist a path from $Start$ to G.

4.3.2 Optimality

GBFS is nonoptimal.

Firstly, completeness is a necessary condition for optimality. Since GBFS is incomplete, therefore it is nonoptimal.

Secondly, we can come up with an example where GBFS returns a sub-optimal solution as a counter example to show that GBFS is nonoptimal.



In the figure above, after setting off from *Start*, the frontier of GBFS will be {*Goal*, *Intermediate*} because $h(Goal) = 0 < h(Intermediate)$, then GBFS will return the path $Start - Goal$, which is longer than $Start - Intermediate - Goal$. This forms a counter example and proves that GBFS is nonoptimal.

4.3.3 Time & space complexity

$O(b^m)$, *where b is the branching factor and m is the maximum depth of the searching tree.*

4.4 Adversarial search

When there are opponents in a game and they change he state of the environment to minimize our agent's utility, the search strategy we use is called adversarial search.

4.4.1 Minimax

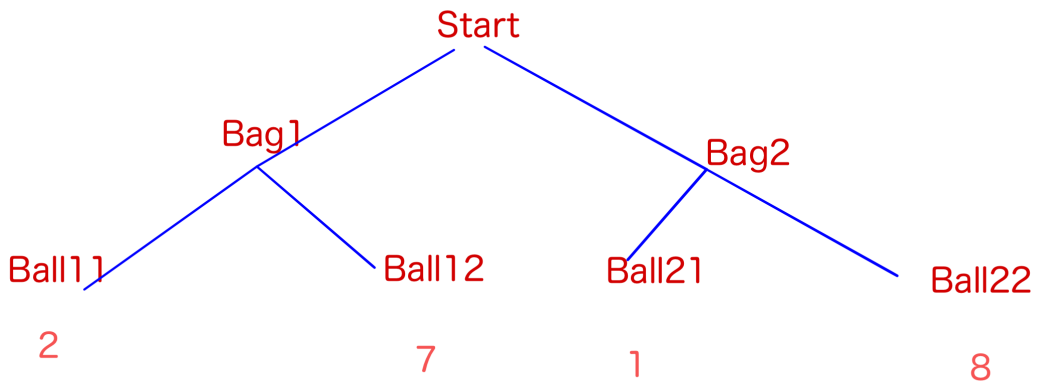
4.4.1.1 Idea of Minimax

A game is stated as following:

There are 2 bags indexed 1 and 2, each contains 2 balls, we use ball 11, 12, 21, 22 to denote the 4 balls. Each ball has a value, ball[11]=2, ball[12]=7, ball[21]=1, ball[22]=8.

There are 2 players, we call them max and min. At first, max chooses a bag, then min chooses a ball from the bag. Max's goal is to maximize the chosen ball's value while min's goal is to minimize it.

We draw a graph as following.



Now, max needs to choose a bag, he would think: if I choose bag[1], then min would choose ball[11] which is 2; if I choose bag[2], then min would choose ball[21] which is 1. In order to maximize the value of the chosen ball, I should choose bag[1].

Let's look at the process again. Max thinks bag[1] is better than bag[2] because $result\ of\ bag[1]) = 2 > result(bag[2]) = 1$, hence the optimal outcome max should expect is $minimax(start) = max(result(bag[1]),\ result(bag[2]) = 2$. Then how did we obtain $result(bag[1])\ and\ bag[2]$? $minimax(bag[1]) = min(ball[11],\ ball[12])$, $minimax(bag[2) = min(ball[21],\ ball[22])$.

4.4.1.2 Abstraction of Minimax

Now we can derive the abstraction of Minimax:

$$Minimax(s) = \begin{cases} \text{Utility(s)}, & \text{if } Terminal - Test(s) \\ max_{a \in Actions(s)} Minimax(Result), & \text{if } Player(s) = Max \\ max_{a \in Actions(s)} Minimax(Result), & \text{if } Player(s) = Min \end{cases}$$

4.4.2 $\alpha - \beta$ pruning

4.4.2.1 Idea of $\alpha - \beta$ pruning

Let's reconsider the example in section 4.4.1. Suppose max has already evaluated ball[11], ball[12], and ball[21], does it really need to evaluate ball[22]?

No. Since $result(bag[2]) = min(ball[21],\ ball[22]) \leq ball[21] = 1 < result(bag[1])$, it is for sure that bag[2] is a worse choice than bag[1], therefore no matter what value ball[22] should be, max need not to evaluate ball[22] at all. Hence ball[22] can be pruned.

When the game tree has multiple layers and large branching factor, pruning an entire branch will significantly reduce the workload of a searching algorithm.

4.4.2.2 Abstraction of $\alpha - \beta$ pruning

$\alpha[n]$ = the highest observed value on path from n, $\alpha[n]$ initially = $+\infty$

$\beta[n]$ = the lowest observed value on path from n, $\beta[n]$ initially = -∞

Pruning:

- given a Min node n, prune the branch below n if there is some Max ancestor i of n such that $\alpha[i] \geq \beta[n]$
- given a Max node n, prune the branch below n if there is some Min ancestor i of n such that $\beta[i] \leq \alpha[n]$