AI SMPS 2023 Week 3 Algorithms

Prepared by S. Baskaran

Node Order

MoveGen determines the order in which nodes are generated, and algorithms determine the order in which nodes are inspected. Some algorithms follow MoveGen order and some reorder the nodes.

For each algorithm discussed in this course, study the order in which nodes are added-to and removed-from OPEN/CLOSED lists and other data structures.

It is important to understand how list structures are constructed, accessed and printed. Study the ‘cons’ and ‘append’ operators and pay attention to the order in which items are cons-ed (added) and appended to lists.

WARNING: while answering auto-graded short-answer type questions, it is important to maintain the algorithm specific node order, otherwise, the auto-grader will mark the answer as wrong and there are no partial marks.

WARNING: often, participants understand the concepts correctly and they may follow a different variation of an algorithm or follow a different implementation of cons and append operators and arrive at a practically viable solution but with a different node order, in this case the auto-grader will mark the answer as wrong.

Tie Breaking a Pool of Best Nodes

MoveGen determines the order in which nodes are generated, and algorithms determine the order in which nodes are inspected. Some algorithms follow MoveGen order and some reorder the nodes.

Cost based (deterministic or heuristic) algorithms select the best node in each iteration. Often there will be a pool of best nodes with the same cost and we do not know which nodes lead to a solution.

In a real world scenario, an additional criteria (external to the algorithm) may be provided to filter the pool of best nodes, or else a random node may be selected from the pool. Each such selection may potentially lead to a different solution or no solution at all.

A pool of best nodes make it difficult to setup all possible answers for an auto-graded question, so we use node label to break ties.

> When multiple nodes have the same (best) cost, sort those nodes in alphabetical order of node label and select nodes from the head of the sorted list.

This tie-breaker helps us to focus on the algorithm and not get beat up on the finer details of a (made up) toy problem.

On the other hand, outside the scope of auto-graded assignments and exams, we urge you to investigate all the problems/solutions/algorithms and find ways to improve it. Wish you the best.

Algorithms

This is a continuation of algorithms published in Week 2 Notes.

```
BEST-FIRST-SEARCH(S)
OPEN ← (S, null, h(S)) : []
CLOSED ← empty list
while OPEN is not empty
    nodePair ← head OPEN
    (N, _, _) ← nodePair
    if GOALTEST(N) = TRUE
        return RECONSTRUCTPATH(nodePair, CLOSED)
    else CLOSED ← nodePair : CLOSED
        neighbours ← MOVEGEN(N)
        newNodes ← REMOVESEEN(neighbours, OPEN, CLOSED)
        newPairs ← MAKEPAIRS(newNodes, N)
        OPEN ← $\text{sort}_h$ ( newPairs ++ tail OPEN )
return empty list
```

```
HILL-CLIMBING(S)
N ← S
do  bestEver ← N
    N ← head $\text{sort}_h$ MOVEGEN(bestEver)
while h(N) is better than h(bestEver)
return bestEver
```

```
VARIABLE-NEIGHBOURHOOD-DESCENT(S)
MoveGenList ← $\text{MOVEGEN}_1$ : $\text{MOVEGEN}_2$ : ⋯ : $\text{MOVEGEN}_n$ : []
bestNode ← S
while MoveGenList is not empty
    bestNode ← HILL-CLIMBING(bestNode, head MoveGenList)
    MoveGenList ← tail MoveGenList
return bestNode
```

```
BEST-NEIGHBOUR-SEARCH(S)
N ← S
bestSeen ← S
until some termination condition
    N ← best MOVEGEN(N)
    if N is better than bestSeen
        bestSeen ← N
return bestSeen
```

```
ITERATED-HILL-CLIMBING(N)
bestNode ← random candidate solution
repeat N times
    currentBest ← HILL-CLIMBING(new random candidate solution)
    if h(currentBest) is better than h(bestNode)
        bestNode ← currentBest
return bestNode
```

Beam Search

We will use the following version of Beam Search in assignments and in the final exam. It exhibits assignment-friendly termination property that avoids infinite loops. Here, **sort$_h$** sorts from best to worst **h**-values.

```
BEAM-SEARCH(S, w)
OPEN ← S : []
N ← S
do  bestEver ← N
    if GOAL-TEST(OPEN) = TRUE
    then return goal from OPEN
    else  neighbours ← MOVE-GEN(OPEN)
          OPEN ← take w ($\text{sort}_h$ neighbours)
          N ← head OPEN    ▷ best in new layer
while h(N) is better than h(bestEver)
return bestEver
```

```
MOVE-GEN(OPEN)
neighbours ← []
for each X in OPEN
    neighbours ← neighbours ++ MOVE-GEN(X)
return neighbours    ▷ the list preserves duplicates
```

(**take** n LIST) returns at most n values from the beginning of LIST.

[o, u, t] = **take** 3 [o, u, t, r, u, n]

[a, t] = **take** 3 [a, t]

[a] = **take** 3 [a]

[] = **take** 3 []

TSP Algorithms

Nearest-Neighbour-Heuristic

```
1  Start at some city
2  Move to the nearest neighbour
       as long as it does not close the loop prematurely
```

Greedy-Heuristic

```
1  Sort the edges by edge-cost
2  Add shortest available edge to the tour
       as long as it does not close the loop prematurely
       and as long as branches/forks are not formed
```

Savings-Heuristic

```
 1  n ← from N cities select a base city
 2  construct (N − 1) sub-tours each of length 2 anchored at n

 3  savingsList ← empty list
 4  for each non base city a
 5      for each non base city b,  if $b \neq a$
 6          savings ← cost(n, a) + cost(n, b) − cost(a, b)
 7          savingsList ← (a, b, savings) : savingsList

 8  sort savingsList in descending order of savings
 9  for each triple (a, b, savings) in savingsList
10      if edges (n, a) and (n, b) belong to two sub-tours
11          merge the two sub-tours:
                remove the edges (n, a) and (n, b)
                insert the edge (a, b)
12  return the tour
```