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 over finer details of (made up) toy problems in the assignments.

In the assignments and exams, when we apply a tie-breaker, we will accept the outcome without further debate/analysis.

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

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 ← sort_h ( newPairs ++ tail OPEN )
return empty list
```

Best-Neighbour-Search(S)

```
N ← S
bestEver ← S
until some termination condition
    N ← best Move-Gen(N)
    if N is better than bestEver
        bestEver ← N
return bestEver
```

Hill-Climbing(S)

```
N ← S
do  bestEver ← N
    N ← head sort_h MoveGen(bestEver)
while h(N) is better than h(bestEver)
return bestEver
```

Variable-Neighbourhood-Descent(S)

```
MoveGenList ← MoveGen_1 : MoveGen_2 : ··· : MoveGen_n : []
bestNode ← S
while MoveGenList is not empty
    bestNode ← Hill-Climbing(bestNode, head MoveGenList)
    MoveGenList ← tail MoveGenList
return bestNode
```

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
```

Random-Walk(N)

```
node ← random candidate solution or start node
bestNode ← node
repeat N times
    node ← random node from MoveGen(node)
    if h(node) is better than h(bestNode)
        bestNode ← node
return bestNode
```

Beam Search

We will use the following version of Beam Search in assignments and exams because it is assignment friendly and prevents infinite loops.

Here, **sort$_\mathbf{h}$** sorts from best to worst **h**-values.

```
BEAM-SEARCH(S, w)
OPEN ← S : []
N ← S
do  bestEver ← N
    if OPEN contains goal node
        return that goal node
    else neighbours ← MOVE-GEN(OPEN)
        OPEN ← take w (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.

$[o, u, t] = \textbf{take}\ 3\ [o, u, t, r, u, n]$

$[a, t] = \textbf{take}\ 3\ [a, t]$

$[a] = \textbf{take}\ 3\ [a]$

$[] = \textbf{take}\ 3\ []$

```
Best-Neighbour-Search(S)
N ← S
bestEver ← S
until some termination condition
    N ← best Move-Gen(N)
    if N is better than bestEver
        bestEver ← N
return bestEver
```

```
Tabu-Search(tt)
F ← array of N zeros                          ▷ use frequency to compute penalty
M ← array of N zeros                          ▷ use memory to track tenure
currentNode ← choose a node randomly          ▷ start node
bestEver ← currentNode

while some termination criteria
    Generate-Neighbours
    Find-Best-Neighbour
    if bestAllowedValue is not worse than eval(currentNode)
        Move-To(bestAllowedIndex)             ▷ improvement
    else if bestValue is better than eval(bestEver)
        Move-To(bestIndex)                    ▷ the aspiration criterion
    else Find-Best-Allowed-With-Penality      ▷ diversify search
        Move-To(bestAllowedIndex)
return bestEver

Generate-Neighbours
    for i ← 1 to N
        neighbour(i) ← Change(currentNode, i)
        value(i) ← eval(neighbour(i))
        tabu(i) ← if M(i) > 0 then YES else NO

Find-Best-Neighbour
    (bestValue, bestIndex) ← (worst value, null)
    (bestAllowedValue, bestAllowedIndex) ← (worst value, null)
    for i ← 1 to N
        if value(i) is better than bestValue
            (bestValue, bestIndex) ← (value(i), i)
        if tabu(i) = NO and value(i) is better than bestAllowedValue
            (bestAllowedValue, bestAllowedIndex) ← (value(i), i)

Find-Best-Allowed-With-Penality
    ▷ use frequency to diversify search
    (bestAllowedValue, bestAllowedIndex) ← (worst value, null)
    for i ← 1 to N,  if tabu(i) = NO
        value(i) ← value(i) − Penalty(F(i))   ▷ diversify search
        if value(i) is better than bestAllowedValue
            (bestAllowedValue, bestAllowedIndex) ← (value(i), i)

Move-To(index)
    currentNode ← neighbour(index)
    if value(index) is better than eval(bestEver)
        bestEver ← currentNode
    for i ← 1 to N,  if M(i) > 0
        M(i) ← M(i) − 1
    M(index) ← tt
    F(index) ← F(index) + 1
```

Random-Walk(N)

```
node ← random candidate solution or start node
bestNode ← node
repeat N times
    node ← random node from MoveGen(node)
    if h(node) is better than h(bestNode)
        bestNode ← node
return bestNode
```

Simulated-Annealing

```
node ← random candidate solution or start node
bestNode ← node
T ← some large value
for time ← 1 to number-of-epochs
    while some termination criteria
        neighbour ← Random-Neighbour(node)
        ΔE ← eval(neighbour) – eval(node)
        if random(0, 1) < 1/(1 + e^(−ΔE/T))
            node ← neighbour
            if eval(node) is better than eval(bestNode)
                bestNode ← node
    T ← Cooling-Function(T, time)
return bestNode
```

TSP Algorithms

Nearest-Neighbour-Heuristic

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

Greedy-Heuristic

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

Savings-Heuristic

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

savingsList ← empty list
for each non base city a
    for each non base city b,  if b ≠ a
        savings ← cost(n, a) + cost(n, b) − cost(a, b)
        savingsList ← (a, b, savings) : savingsList

sort savingsList in descending order of savings
for each tuple (a, b, _) in savingsList
    if edges (n, a) and (n, b) are in different tours
        merge those tours into a single tour:
            remove the edges (n, a) and (n, b)
            and insert the edge (a, b)
return the final tour
```