Implementing A-Star as special case formulation of General Branch and Bound using Missionary and Cannibal Problem

A PROJECT REPORT

Submitted in partial fulfilment of the requirement of the course

19AI602

Foundations of Artificial Intelligence

by

Name	Roll No
Animesh Deb	**(AM.EN.P2ARI20006)**
Dhanyalaxmi Panickar	**(AM.EN.P2ARI20012)**
Kaushik S	**(AM.EN.P2ARI20017)**

Under the Guidance of

Dr. Georg Gutjahr

(Assistant Professor)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMRITA VISHWAVIDYAPEETHAM

KOLLAM-690525, KERALA, INDIA

JANUARY 2021

Contents

Chapter 1

Introduction

Branch and Bound is a systematic method for solving combinatory, discrete, and general mathematical optimization problems, a rather general optimization technique that can be useful in finding the solution when the greedy method and dynamic programming fail to provide a solution.

The basic idea is that for a given problem the Branch-and-Bound algorithm explores the entire search space of possible solutions and provides an optimal solution. The problems like finding the shortest path in a graph, minimum spanning tree, production planning, and crew scheduling are some examples of the problem which have a finite but extensive number of feasible solutions. Therefore, the Branch-and-Bound algorithm can effectively solve such problems.

We will demonstrate the concept used in our project through the medium of the Missionaries and Cannibals problem, a classic AI puzzle. It can be defined as:

On one bank of a river are three missionaries and three cannibals. Only one boat available that can hold up to two people. If the cannibals ever outnumber the missionaries on either of the riverbanks, the missionaries will be eaten. The boat cannot cross the river by itself with no people on board. With these constraints, all the three missionaries and three cannibals must cross a river using the boat.

Chapter 2

Objective

The objective of our project is to prove conceptually and with a demonstration that A-star is a special case of general Branch-and-bound.

The principle of Branch-and-Bound is that the total set of feasible solutions can be partitioned into smaller subsets (which are represented by the nodes of the branching tree) of solutions, and these smaller subsets can be evaluated systematically until the best solution is found. Therefore, we improve on this by adding a heuristic value to each subset concerning the goal that is how far the subset is from reaching the status of the goal. This will give the partitioning a directionality towards the goal, which will result in a much more efficient algorithm.

Chapter 3

Algorithms

3.1 Branch and bound

Branch-and-bound methods are methods based on a clever enumeration of the possible solutions of a combinatorial optimization problem. The principle consists of partitioning the solution space into disjoint subsets, which are represented by the nodes of the branching tree. Then, the algorithm explores the branches of the tree according to a routing strategy. To avoid exploring the entire tree, before creating a new node in the tree, the algorithm evaluates the node by comparing the value of the best possible solution, which could be found in the corresponding sub tree, with the best current solution. If a better solution cannot belong to the sub tree rooted at the considered node, the sub tree is discarded. Otherwise, the exploration process continues. This method can be seen as a divide-and-conquer approach. Lots of combinatorial optimization problems may be solved, more or less efficiently, using a branch and bound strategy.

3.2 Best-First Search

Best-First Search is a blind variant of Branch and Bound in which future costs are all assumed to be equal. When a solution is found, it is the cheapest. Both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent most is promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

3.3 A-Star Search

A* is like **Greedy Best-First-Search** in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search. The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favouring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal. For finding the next move it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$.

Chapter 4

Project Concept

Branch and Bound is a problem-solving technique which is widely used for various problems encountered in operations research and combinatorial mathematics. Various heuristic search procedures used in artificial intelligence are related to B&B procedures. Our project is based on the concept of the paper *Nau, Dana S.; Kumar, Vipin; Kanal, Laveen (1984). "General branch and bound, and its relation to A* and AO*" doi:10.1016/0004-3702(84)90004-3 '* which presents a formulation of B&B general enough to include A* formulation as special case, and shows how this well-known AI search procedure is a special case of this general formulation. Branch and Bound can be abstrtactly stated in the following form:

Given a (possibly infinite) discrete set X and a real-valued objective function F whose domain is X, find an optimal element x* E X such that

$F(x^*) = \min\{F(x) \mid x \in X\}$

4.1 General Formulation of Branch and Bound

Shown below is the procedural schema (P0),which describes the basic concept of Branch and Bound.

4.1.1.Procedure P0

1. ACT := {X} *//ACT is the current active set//*

2. loop *//the main loop//*

3. if ACT = {Z} for some Z and Z is a singleton {z} then

4. return z

5. endif

6. SEL := select (ACT)

//select some of the sets in ACT//

7. SPL := split(SEL) *//split the sets in SPL//*

8. ACT := prune((ACT- SEL) u SPL)

//remove the selected sets from ACT, replace//

//them by the newly generated sets, and then//

//prune unneeded sets from ACT//

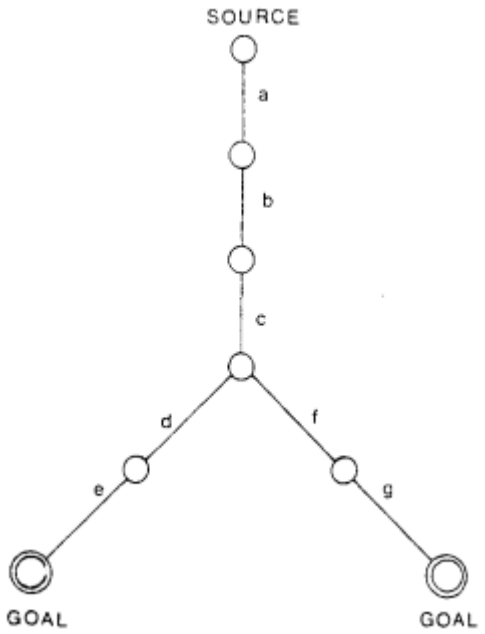
9. repeat

end P0

In the above procedure, ACT refers to the active set, which is a collection of subsets of X. SEL, the selection function, is any function which returns a collection SEL, a subset of ACT. The domain of select is the set of all possible values which ACT might have at line 6 of P0. split, the splitting function, has the set of all possible values which the collection SEL might have at line 7 of P0 as its domain . split(SEL) returns a collection SPL of subsets of X such that the sets in SPL contain precisely those elements which are members of SEL. prune, the pruning function, has as its domain the set of all possible values which the collection of sets R = (ACT-SEL) U SPL might have at line 8 of P0. prune returns a collection of sets R' is subset of R such that at least one of the minimum elements of R is also present in R'.

4.2 Inadequacies of P0

As an example, consider the below shown directed graph G for the least-cost path problem- The problem of finding a path from a source node s in G to any member of a set T of terminal nodes. The set X of solutions to this problem is the set containing each path from s to any member of T. The least-cost path problem is the problem of finding a path P in X which minimizes the value of the objective function cost (P).



G is the directed graph having node set {a, b, c, d, e, f, g, h} and arc set

{(a, b), (b, c), (c, d), (d, e), (c, f), (f, g), (g, h)}, and let s = a and T = {e, g}.

If split were defined as it is in P0 i.e as a function of the sets represented . The paths (a) and (a, b) both represent {(a, b, c, d, e), (a, b, c, f, g)}; (a, b, c, d) represents {(a, b, c, d, e)}; and (a, b, c, f) represents {(a, b, c, f, g)}. If split is defined as P0 then

split({{(a, b, c, d, e), (a, b, c, f, g, h)}}) = {{(a, b, c, d, e), (a, b, c,f, g, h)}}

split({{(a, b, c, d, e), (a, b, c, f, g, h)}}) = {{(a, b, c, d, e)}, {(a, b, c, f, g, h)}}

Since both the above split assign two different values to split({{(a, b, c, d, e), (a, b, c,f, g, h)}}), split is ill-defined when considered as a function of the sets represented rather than their representations. This demonstrates that P0 is inadequate to describe the behavior of the splitting functions used in some B&B procedures. This example also illustrates that the termination test used in P0 does not adequately model the termination tests used in practical B&B procedures.

4.3 An improved model of branch and bound

Inadequacies of P0 can be eliminate by defining an explicit goal function.In P0 even after reaching the goal node, P0 went up to node h. Since, it didn't have a goal function. Representation scheme is defined as a pair (S, rf), where S is a set of representations and rf is a representation function. In this report, r denotes a representation and R denotes a collection of representations. For convenience, we define,

$$Fmin(r) = \min \{F(x) \mid x \in rf(r)\}$$

And

$$Fmin(R) = \min \{F(x) \mid x \in rf(r) \text{ for some } r \in R\}$$

With this enhancement instead of splitting the whole path, we get a method to specifically move to required node and perform the goal test and thus enabling us to reconstruct the path as required.

4.3.1.Procedure P2

GBB with an auxiliary database (DB2)//

1. Initialize DB2

//DB2 consists of all auxiliary information used by P2//

2. ACT2:= {r0}

//rO is the initial representation of X//

3. loop *//the main loop//*
4. if ACT2 = {r2} for some r2 and goal2(r2,DB2) holds then
5. return r2
6. endif
7. SEL.2 := select2(ACT2,DB2)
8. (SPL2,DB2) := split2(SEL2,DB2)
9. (ACT2,DB2) := prune2(ACT2- SEL2,SPL2,DB2)
10. repeat

end P2

Explained below are the fundamental properties of P2 :

Let t be the number of times the main loop is fully executed before a return occurs.

For ith iteration such that, $0 \leq i < t + 1$

$$ACT2^{i+1} = prune2((ACT2^{i} - SEL2^{i+1}), SPL2^{i+1})$$

From which we can proof

$$Fmin(ACT2^{i+1}) = Fmin(ACT2^{i})$$

From where we can say,

For every integer i such that $0 \leq i < t + 1$,

$$Fmin(ACT2^{i}) = \min\{F(x) \mid x \in X\},$$

4.4 Special Case P3

To improve the performance and for ease in pruning, many B&B procedures maintain a record of the 'best solution seen so far' separately from the active list. Such procedures usually are instances of the procedure P3 below.

4.4.1.Procedure P3:

//a special case of GBB//

1. BEST3 := 'unknown'

//we define rf3('unknown') to be 1~,//

//whence Frn~('unknown') = oo//

2. ACT3 := {r0>}

//rO is the initial representation of X//

3. loop

4. if ACT3 = 0 then return BEST3 endif

5. SEL3 := select3(ACT3)

6. if SEL3 is a singleton {r3} and goal3(r3) then

7. if Fmin(r3) < Fmin(BEST3) then

8. BEST3 := r3

9. endif

10. else

11. SPL3 := split3(SEL3)

12. ACT3 := prune3(ACT3- SEL3,SPL3)

13. endif

14. repeat

end P3

With P3 the performance improved, but still P3 doesn't have sense for the direction of goal. This is achieved in the A* algorithm.

4.5 A*

We can solve the least-cost path problem using A* procedure , which appears below as procedure P6 .In P6, c(n, n'); g(m) is the cost of the arc (n, n'); g(m) is the cost of the least-cost path P seen so far from the source node s to m; and

P = path(m) is the path (s ..... parent(parent(n)), parent(n), n). h(m)>i 0 is a lower bound on the cost of any path from m to a member of T, and

f(n)= g(n)+ h(n) is a lower bound on the cost of extending P to a member of T.

4.5.1.Procedure P6: //A*//

2. OPEN := list containing the source node s
3. CLOSED := NIL
4. while OPEN ≠ NIL do *//the main loop//*
5. n := removetop(OPEN) *//remove first element//*
6. insert n into CLOSED
7. if n ∈ T then
8. return path(n)
   //path(n) is the path//
   //(s ..... parent(parent(n)), parent(n), n)//
9. else
10. for every child n' of n in G do
    //compute g (n') and f(n')//
11. gg := g(n) + c(n,n')
12. if:= gg + h(n')
13. for all nodes m in OPEN or CLOSED do
14. if m = n' and f(m) ≤ ff then
15. goto PRUNE *//prune n'//*
16. else if m = n' and if< f(m) then
17. call remove6(m)
18. endif
19. endfor
20. parent(n') := n
21. g(n') := gg
22. f(n') := ff
23. OPEN := insert6(n',OPEN)
    //insert n' into OPEN just after the last//
    //node n such that f(n) < f(n')//

24. PRUNE: endfor
25. endif
26. endwhile
27. return 'unknown'
end P6

In P6,
*procedure remove6(*m)
1. if m ∈ OPEN then remove m from OPEN endif
2. if m ∈ CLOSED then remove m from CLOSED endif
3. for every n such that parent(n) = m do
4. call romove6(n)
5. endfor
end remove6

Each node n in OPEN or CLOSED represents the path -
path(n) = (s ..... parent(parent(n)), parent(n), n).
In order to make this representation explicit, P6 is rewritten as procedure P7 below.

4.5.2.Procedure P7: //A*, rewritten//

```
2. ACT7 := list containing the null path from s to s
3. GEN7 := NIL
4. while ACT7 ≠ NIL do //the main loop//
5.      {P} := select7(ACT7)    //select first member P of ACT7//
6.       insert P into GEN7
7.       if goal7(P) then
8.            return P
9.      else
10.          SPL7 := split7({P})
11.          ACT7 := ACT7 - {P}
12.          for every path P' in SPL7 do
13.             for every Q in ACT7 or GEN7 do
14.                 if tip(Q) = tip(P') and L7(Q) ~< L7(P') then
15.                    goto PRUNE    //prune P'//
16.                 else if tip(Q) = tip(P') and L7(P') < L7(Q) then
```

17. call remove7(Q)

18. endif

19. endfor

20. predecessor(P') := P

Let m be any node generated by P6, and let P= path(m), where tip(P) = m. To write P7, the following definitions are used.

(1) rf7(P) is the set of all extensions of P to members of the set of terminal nodes T (i.e., the set of all paths PQ such that Q is a path from tip(P) to a member of T and the only member of T in Q is tip(Q)). Thus rf7 is a representation function.

(2) goal7(P) holds if and only if tip(P) ∈ T.

(3) select7(L) returns a set whose only element is the first element of the list L. Since the list ACT7 is always kept ordered according to the lower bounds of its members, select7 corresponds to remove top in P6.

(4) split7(P)= {Pn | n is a child of tip(P)}. Expanding a node m in P6 corresponds to computing split({path(m)}) in P7.

(5) L7(P)= cost(P)+ h(tip(P)). Thus from the definitions of f, g, and h,

L7(P) = f(tip(P)), whence L7(P) is a lower bound on Fmin(P).

(6) insert7(P, L) inserts P into the list L just after the last path Q in L such that L7(Q)< L7(P). Thus select7 is an L7-best-first selection function.

Using these definitions, it is clear that P6 can be rewritten as procedure P7.

4.6 A* a special case of GBB

Below it is shown that P7 (and hence A*) is a special case of GBB.

Let G be a graph for a shortest path problem with start node s and goal set

T, and let P and Q be paths in G from s.

Then P covers Q if F_{min}.(P) ≤ F_{min} (Q) and there are paths P' and P'' such that P = P'P'' and tip(P') = tip(Q). If P covers Q, this means that for every extension Q* of Q to a member of T, there is an extension P* of P to a member of T such that

F_{min} (P*) ≤ F_{min}(O*),

hence Q can be pruned if Q ≠ P and P ∈ ACT7. Note that the covering relation is transitive: if P covers Q and Q covers R, then P covers R.

Lemma C.1. If P prunes Q in lines 15 or 17 of P7, then P covers Q.

Proof. If P prunes Q, then tip(P) = tip(Q) and LT(P) ≤ LT(Q). But

L7(P) = cost(P) + h(P) and L7(Q) = cost(Q)+ h (Q)= cost(Q)+ h (P),

So,

cost(P) ≤ cost(Q).

Since tip(P) = tip(Q), this means that F_{min} (P) ≤ F_{min} (Q), hence P covers Q.

Let t be the number of times the main loop of P7 is fully executed for some input graph G (thus, t may be infinite). Let k(i) be the number of times lines 12-22 of P7 are executed during the ith iteration of the main loop. Note that k (i) = |split7(P^i)|.

We define (i, j) to be a loop index of P7 if it corresponds to the jth iteration of the inner loop of P7 during the ith iteration of the main loop; i.e., if 0≤i<t+l and 0 ≤ j ≤ i k(i). We say that (i',j') is older than (i,j) if (i',j') corresponds to an iteration of the inner and main loops of P7 previous to the iteration corresponding to (i,j); i.e., if i'<i or if i'= i and j' <j.

Let ACT7^{i,j} and GEN7^{i,j}, respectively, be the values of ACT7 and GEN7 computed on the jth iteration of lines 12-22 of P7 during the ith iteration of the main loop. Let $P_{i,1}$, $P_{i,2}$..... $P_{i,k(i)}$ be the members of SPL7^i computed at line

10 of P7, and let

frontier(i, j) = ACT7^{i,j} ∪{ $P_{i,j+1}$, ..., $P_{i,k(i)}$}.

Note that for each i,

(1) ACT7^{i,0} = ACT7^{i-1} - {P_i} ;

(2) GEN7^{i,0} = GEN7^{i-1} ;

(3) frontier(i, 0) = (ACT7^{i-1} - {P_i}) ∪ SPL7^i ;

(4) GEN7$^{i,k(i)}$ = GEN7^i ,

(5) frontier(i, k(i))= ACT7$^{i,k(i)}$ = ACT7^i.

Theorem C.2. For every loop index (i, j), every loop index (i', j') older than (i, j), and every path V E frontier(i', j'), there is a path W E frontier(i, j) such that W covers V.

Proof : (by induction on (i,j)). There is no (i', j') older than (1,0), so the theorem holds vacuously for (i,j)= (1,0). For the induction hypothesis, let

0≤i<t+l and 0 ≤ j ≤ i k(i), and suppose that the theorem holds for every (i'', j'') older than (i, j). To prove that the theorem holds for (i, j), there are two possible cases to consider: j = 0 and j > 0.

CASE 1: j = O. Let (i', j') be older than (i, 0). If i' < i - 1 or if i' = i - 1 and

j'< k(i-1), then by the induction hypothesis every V ∈ frontier(i', j') is covered by a W∈ frontier(i - 1, k(i -1)). Thus since covering is transitive, it suffices to show that every V ∈ frontier(i-1, k(i-1)) is covered by a w ∈ frontier (i, 0). If

V ∈ frontier (i - 1, k (i - 1)), then either V ∈ ACT7^{i,0} or V = P^i.

In the first case, V covers itself. In the second case, it follows from the definition of split7 that

F_{min} ($\{P_{i,1}, P_{i,2}..... P_{i,k(i)}\}$) = F_{min} (P^i), whence one of $P_{i,1}, P_{i,2}..... P_{i,k(i)}$ covers P^i. In either case, the theorem holds for (i, j).

CASE 2: j >0. Let (i', j') be older than (i,j). If i'< i or if i' = i and j' <j - 1, then by the induction hypothesis every V E frontier(i',j') is covered by a

W ∈ frontier(i, j - 1). Thus since covering is transitive, it suffices to show that

every V ∈ frontier(i, j-1) is covered by a W ∈ frontier(i, j). There are three possible cases to consider.

Case 2(a): V ∈ $\{P_{i,j+1}, P_{i,2}..... P_{i,k(i)}\}$. Then V ∈ frontier(i, j) and V covers V.

Case 2(b): V = $P_{i,j}$. If V ∈ $\mathrm{ACT7}^{i,j}$, then V ∈ frontier(i, j) and V covers V.

Otherwise, V was pruned at line 15 of P7 by some a Q ∈{$\mathrm{ACT7}^{i,j-1} \cup \mathrm{GEN7}^{i,j-1}$.By Lemma C.1, Q covers V, and by the induction hypothesis, there is a W ∈ frontier (i, j-1) covering Q, whence W covers V. Since V could not have pruned W,

W ∈ $\mathrm{ACT}^{i,j} \subseteq$ frontier (i, j).

Case 2(c): V ∈ $\mathrm{ACT7}^{i,j-1}$. If V ∈ $\mathrm{ACT7}^{i,j}$, then V covers V. Otherwise, V was pruned at line 17 of P7 by $P_{i,j}$, whence from Lemma C.1, $P_{i,j}$ covers V.

Thus, since covering is transitive, this case reduces to Case 2(b).

Corollary C.2.1. P7 is an instance of P3B.

Proof : It is clear that goal7 is a goal function, and that select7 and split7 have the properties of P3-selection and -splitting functions, and that select7 is L7-best-first. If we define prune7 to be lines 11-22 of P7, it follows from Theorem C.2 it follows that prune7 has the properties of a P3-pruning function. It only remains to show that select7 split7, and prune7 are defined for all arguments, which might be given to them during the operation of P7, and this is easily proved by induction. From Corollary C.2.1, it is clear that P7 (and hence A*) is a special case of GBB.

Chapter 5

State Space

State Space: It includes all possible states (triples) at the initial side of the river possible with given constraints.

triple (#missionaries, #cannibals, #boat)
where,

$0 \leq$ #missionaries ≤ 3
$0 \leq$ #cannibals ≤ 3
$0 \leq$ #boat ≤ 1

Successor function:

- **Actions:**
  - Boat takes 1 missionary across the river.
  - Boat takes 1 cannibal across the river.
  - Boat takes 2 missionaries across the river.
  - Boat takes 2 cannibals across the river.
  - Boat takes 1 missionary and 1 cannibal across the river.
- **Cost:** Number of actions performed
- **Constraints:** Additionally, we need to ensure #cannibals $\leq$ #missionaries on both sides.

Start State: (3, 3, 1)

Goal Test: Is state (0, 0, 0)

Chapter 6

Project Implementation

The project was implemented using primarily 2 classes.

1. class State.
2. class Search.

6.1 class State

This class generates the states for our problem. The state representation is < numMissionary ,numMissionary , boatPresent, numSteps , maxMis, maxCan, isBFS, isAstar >.

Functions in this class:

6.1.1.Function goalTest:

This function tests for our goal, which is to have all the missionaries and cannibals on the other side/ final side of the river. Returns true if the goal has been achieved or false if the state is not a goal state.

6.1.2.Function isValid:

This function checks for the validity of the states provided to it based on our constraint. It verifies that the number of missionaries is greater than the number of cannibals on both sides of the river. Returns true if the state is valid else false.

6.1.3.Function genChild:

This function generates the child states for the state provided to it based on the below-mentioned conditions and returns a list of valid child states generated.

I. The provided state is a valid state, which is validated using the IsValid function.
II. At a time maximum, only two people can be on the boat.
III. The boat cannot go to the other side on its own.

6.2 class Search

Search class performance the operation of finding the solution for the given problem data. During the initialization of the class, which takes four parameters, which are, maximum number of Missionaries, maximum number of Cannibals, search type and whether or not the record for explored states should be maintained or not.

Using this class search for the solution can be completed in three different ways they are Breadth-first search, Branch and Bound search and A-Star search.

The important variables and functions are:

6.2.1. Queue search_queue:

Contains states to be explored in the future.

6.2.2. List search_record:

Keeps record of all the states explored. Used for reconstructing path.

6.2.3.Int search_counter:

Number of states explored so far. In any point in search, ideally search_counter is equal to length of search_record.

6.2.4.recordKeeper:

It is used if record of explored states is to be maintained.

6.2.5. List path:

Used for showing reconstructed path.

6.2.6. Function search:

This function searches through the search space as per search method chosen.

6.2.7. Function showPath:

This function can only be used after successful search. Using this function, the path is constructed back from problem node to goal node.

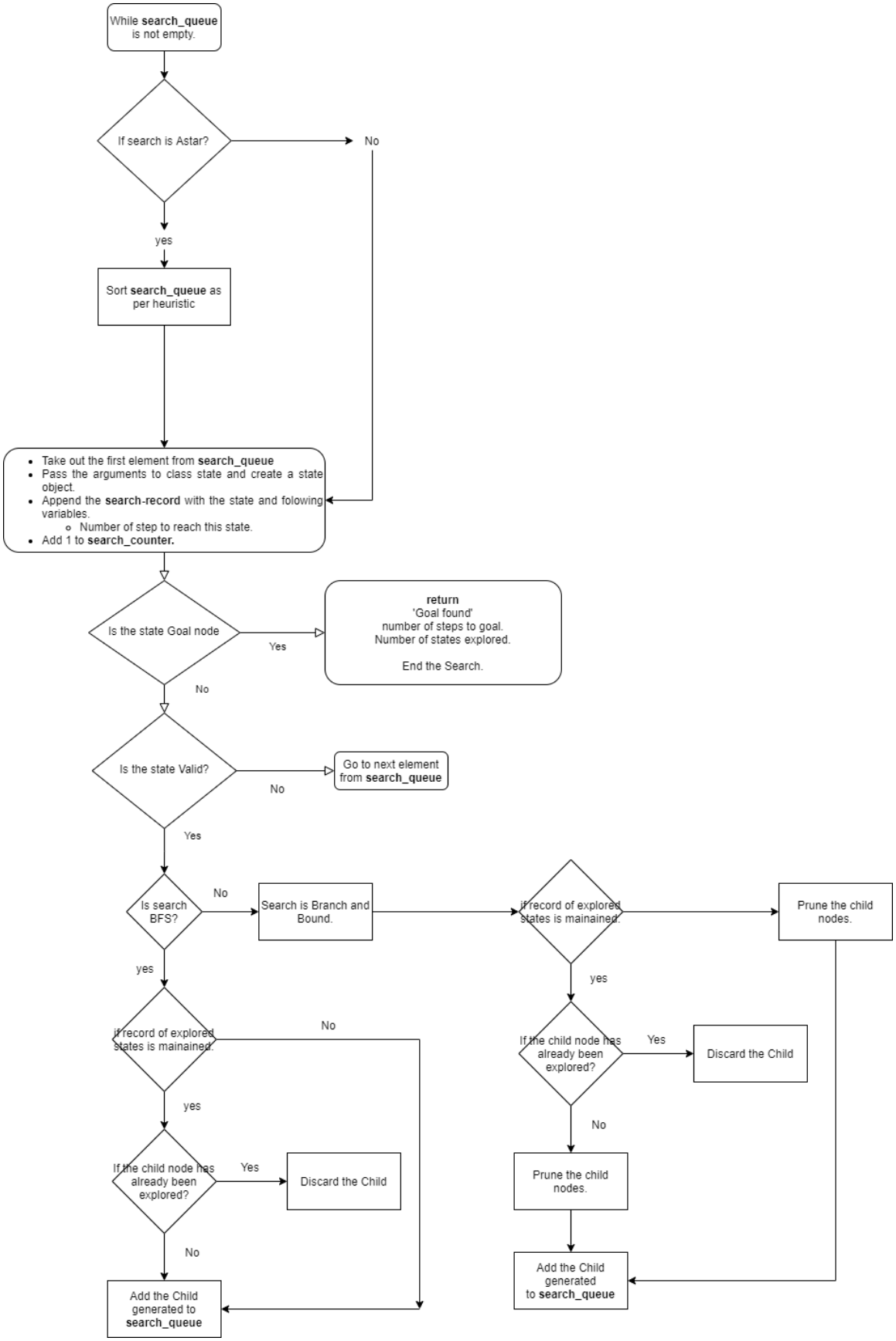


Figure 6.1 Flowchart diagram of search strategy.

Chapter 7

Search Parameters

7.1 Branch and Bound

For the given problem, performing a BFS will maintain the lower bound of cost specified in Chapter 5. The upper bound was calculated as follows:

Upper Bound = Maximum number of Missionaries + Maximum number of Cannibals

+ int(number of steps taken to reach current state/2)

A child was pruned if for the child,

Number of Missionaries + Number of Cannibals - 1 > Upper Bound.

7.2 A Star

For the given problem,

g(x): Number of steps.

h(x): Number of Missionaries + Number of Cannibals – 1

f(x): g(x) + h(x)

Chapter 8

Observation and Conclusion

For 3 missionaries and 3 cannibals, Breadth first search explores 16,469 states, Branch and Bound algorithm explores 272 states and A-Star explores 268 states to give us the solution. The solution obtained is same for all the algorithms and is an optimal solution.

The improvement from Branch and Bound to A-Star does not seem to be quite significant for problem with 3 Missionaries and 3 Cannibals. However, when we increase the number to 4 missionaries and 3 Cannibals, the difference is significance. The same search is completed after exploring 70483 states in BFS, 1706 states in B&B and 1096 states in A*.