

Parallelization of the A* Search

Problem Specification:

We are interested in the problem of creating a parallel version of the A* Search. The A* search is an informed best-first graph search which finds the least-cost path from a given start node to one (of possibly many) goal node.

The cost function is defined as $f(x) = g(x) + h(x)$ where

$g(x)$: is the actual cost from the start node to x . $g(x) \geq 0$

$h(x)$: is an estimated cost from x to any goal node. $h(x) \geq 0$

The first goal node visited in the A* search can be proven to be the optimal path iff the estimated cost $h(x)$ is always less than or equal to the actual cost from x to any goal node.

Sequential Algorithm:

The A* search uses a priority queue (a queue which is sorted such that the least cost element is placed at the front of the queue) to store all the path segments being analyzed. At each iteration, an element is taken from the front of the queue, expanded, and the resulting elements are placed back in the queue. The algorithm stops when an element at the front of the queue is a goal state.

```
function a_star(start_state, goal_states)
    start_state.g_cost := 0
    start_state.h_cost := get_h_cost(start_state, goal_states)
    priority_queue.insert(start_state)      % Sorted by g_cost+ h_cost
    goal_found := False
    while(not goal_found and priority_queue.is_not_empty() )
        current_state := priority_queue.remove_first()
        goal_found := current_state in goal_states
        if(not goal_found)
            foreach child_state in get_child_states(current_state)
                child_state.g_cost := current_state.g_cost +
                    dist_between(current_state, child_state)
                child_state.h_cost := get_h_cost(child_state, goal_states)
                priority_queue.insert(child_state)
    if(goal_found)
        return current_state
    else
        return Failure
```

```
function get_h_cost(current_state, goal_states)
    % This is a heuristic function based on your application to estimate a lower
    % bound on the cost to go from the current_state to any one of the goal_states
```

```
Function get_child_states(current_state)
    % This function expands the current state and returns a set of child states.
```

Example Problem:

Forrest Gump one day decided that he wanted to run, and run he did. He ran all the way across the continental United States several times over, and never between the same two cities twice. At the end of his run, he came up with a list of numbers. Each number in this list corresponded to the time it took

him to go from one city to the next. And Forrest being Forrest does not remember what those cities are, other than that he started from Montgomery Alabama, and ended up in the same place.

Ex:

Montgomery - 2nd City : 10 days
 2nd City – 3rd City : 8 days
 3rd City – 4th City : 21 days
 4th City – 5th City : 6 days
 5th City – Montgomery : 19 days

Your task is to find the names of the cities that Forrest visited, in that order. For this purpose you will have a table containing the running times between neighboring cities. Keep in mind that these times are the average running times between those cities, and may not be the same (although roughly similar) as Forrest’s running times.

Ex:

	Montgomery	Dothan	Columbus	Augusta	Atlanta	Nashville	Birmingham
Montgomery	0	12	8		23		10
Dothan	12	0	7				
Columbus	8	7	0	19	10		
Augusta			19	0	5	30	
Atlanta	23		10	5	0	22	15
Nashville				30	22	0	16
Birmingham	10				15	16	0

Example Solution:

For this example problem, we can utilize the A* search in order to solve the problem.

- The state of the system will be the list of cities visited
- The start state will be {Montgomery}
- The goal states will be all the lists of length 6, where the first and last elements are Montgomery, and there exists a path between neighboring cities in the list.
 Ex: {Montgomery, Dothan, Columbus, Augusta, Atlanta, Montgomery}
- Node expansion will be done by checking whether the last city in the current state has a path to any city in the list of cities.

- The g_cost function will be the difference of distances between Forrest's path P , and the predicted path X .

$$g_cost(X) = \sum_{i=1}^{len(X)-1} Abs(C_P(i-1, i) - C_X(i-1, i))$$

Ex: g_cost ({Montgomery, Dothan, Columbus})

$$\begin{aligned} &= Abs(C_X(\text{Montgomery, Dothan}) - C_P(\text{Montgomery, 2}^{nd} \text{ City})) + Abs(C_X(\text{Montgomery, Dothan}) - C_P(\text{2}^{nd} \text{ City, 3}^{rd} \text{ City})) \\ &= Abs(12 - 10) + Abs(7 - 8) \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

- The h_cost function will be the best possible path that can be assigned for the unpredicted segment of Forrest's path. (ie. The sum of the costs of all the path segments that have the values most similar to Forrest's path)

$$h_cost(X) = \sum_{c=Cost(X)}^{len(P)-1} \min_{j,k} (Abs(C_P(i-1, i) - C_T(j, k)))$$

Here T is the table containing all the distances between cities

Ex: g_cost ({Montgomery, Dothan, Columbus, Augusta})

$$\begin{aligned} &= \text{Min} (Abs(C_P(\text{4}^{th} \text{ City, 5}^{th} \text{ City}) - C_T(j, k))) + \text{Min} (Abs(C_P(\text{5}^{th} \text{ City, Montgomery}) - C_T(j, k))) \\ &= \text{Min} (Abs(6 - C_T(j, k))) + \text{Min} (Abs(19 - C_T(j, k))) \\ &= Abs(6 - 7) + Abs(19 - 19) \\ &= 1 + 0 \\ &= 1 \end{aligned}$$

Ideas for Converting into a Parallel algorithm:

We propose parallelizing this by assigning the node-expansion, and cost calculation steps to individual cores, while keeping the priority queue as shared memory.

- The priority queue is shared among each thread, and must be implemented in a thread-safe fashion.
- Each worker thread is responsible for the following:
 - taking the first element in the priority queue
 - expanding it to find the child states,
 - assigning the g_costs and h_costs to the child states
 - putting the child states back into the priority queue

The terminating criteria are:

- A worker thread W_i finds a goal state in the front of the priority queue.
- All worker threads that started before W_i have put their results back into the priority queue
- The element that W_i picked from the front of the priority queue (in condition 1) has a cost less than the front of the priority queue after condition 2.