Subproblem LSA - 2-d Node List / 3-d Space-Time Map

When I first brainstormed an algorithm using graph traversal that would solve the subproblem, I imagined we needed a grid of i, t nodes rather than just location nodes, as we were able to use in the set-partitioning without column generation (i.e. enumerating all the routes). I said so because otherwise, how could we represent t? But after thinking about it I observed the following:

1. We have a parameter T which we use in the LSA which can capture 'time of entry into last node so far', so we might be able to pull off the subproblem LSA with just this entry to add all the μ_{it} from the reduced cost $C^q - \sum_{i \in \mathcal{I}} \pi_i u_i^q - \rho - \sum_{i \in \mathcal{I}} \sum_{t \in \mathcal{T}} \mu_{it} \delta_{it}^q$;
2. If it is true that in $C^q - \sum_{i \in \mathcal{I}} \pi_i u_i^q - \rho - \sum_{i \in \mathcal{I}} \sum_{t \in \mathcal{T}} \mu_{it} \delta_{it}^q$ we only actually subtract the μ_{it} once*; namely, when we enter node i at the special time t and NOT the subsequent t's for which we *stay* at node i but do not actually move;
3. A 1-d node list / 2-d location map would be more time-efficient than the i, t grid because it would involve far less nodes (whether the bottleneck is in the NDom check or the generation and modification of new-node-searches I will determine soon);
4. If it is true that we actually have to subtract the μ_{it} multiple times*, then given our parameter T, we would be forced to 'retroactively' update all the μ_{it}'s because you'd only know when you leave that node by updating the value of T when you actually stayed at the old node. In particular, this is given by: start index old_value_of_T, end index new_value_of_T minus transportation_time_from_old_node_to_new_node. It is at least doable, but clumsy.

* Do we subtract μ_{it} once for every t at which we are at node i, or only when we enter node i?

I've created and tested a 1-d node list / 2-d location map subproblem LSA and it seems to work fine for the first iteration - it clearly generates multiple feasible routes with negative reduced cost. I'd like to experiment and create a 2-d node list / 3-d space-time map to see how it works! It may actually be a benefit to do so for two reasons:

1. An i, t grid reduces the clumsiness of working with time as a variable; instead, time is embedded within the grid itself;
2. In the ICT document, section 9 "Attempt at a new version" expresses interest in solving the subproblem in which a route embodies not just the location, but also whether work is performed there or not. My i, t-grid solution will come one step closer to making this happen. It's not exactly this solution, because I don't explicitly tell you when to work there, but it makes all the times obvious.

The only true change of this 3d LSA is that our nodes are actually a 2d matrix, not a 1d list. The first index is location, ranging from 1 (depot) to 2 to n+1 (jobs) to n+2 (depot end). The second is time, ranging from 1 (start) to the end of time. The parameters, strategy, etc. are the same. It's just that now, nodes will involve two parameters rather than one. This is more a coding exercise and being careful than anything else.

Differences
When we initialize N, the list of paths, we start with [1, 1] to indicate the first 1 is the depot, and the second 1 is that we started here at time 1.

There will be no more T list. That was used in the 2d case to watch out for incompatible times. In this case, the time information is exactly encoded in [i, t] so if we are not ready for a certain time this will be directly put in t.

Reduced costs R are as they stand.

L represents the last node: keep it like this. Now we initialize with [1, 1] not 1.

And A is as it stands.

Step 1 In the 2d version we only check that, if the location has reached the end depot or the current state is not worth pursuing, we increment the current_state. In this 3d version, we also check that, if we've gone overtime or have reached max time, stop going and move on to the next one.

Step 2 Rather than iterate over a for loop over locations i, we also iterate over a for loop over times t. So this will become more complicated.

However, we should not start with time at 1. Look at the last time we currently have. Extract it with L[current_state][2]. Since we can't go below it, we start with that plus one instead of 1, and go up to global_time.

Step 2-1 In the check for whether a time-space node (tsn) is in the list, we know t has no problem. Instead we check we don't visit a node we visited before.

Step 2-2 Now t indicates exactly when we intend to arrive at the destination. We make the following checks:

1. We have to arrive to have enough time to work. Specifically, t + new_job_time must not exceed new_job_window_close. t + new_job_time <= new_job_window_close.
2. We have to arrive late enough to be actually within the window. Specifically, we require new_job_window_start <= t <= new_job_window_close. The second half is actually redundant because of part 1, because if part 1, then second half. So we only need new_job_window_start <= t.
3. We have sufficient time to arrive from the latest time to the current time. Specifically, L[current_state][2] + travel_time(L[current_state][1], new_node) must be less than or equal to t.
4. Don't just go running away from a previous job. In other words, check that you've spent at least old_job_load_time since the last hit time until you get to this time. The formulation is L[current_state][2] + old_job_load_time + dist <= t.

We can waste as much time as we want in between so 3 is a one-way check and it's good enough.

Step 2-3 For N, copy the original state, then add in the new tsn.
No need to do time!

For cost: well you have i and t right there.
Minor edits ensue.

Note: Performance is very slow. This is the main problem...and the crucial one. In fact, the increase in time dramatically (probably exponentially) increases complexity of the algorithm. There's no way this thing will scale.