1 Planning with Cost

1.1 Transition-based RRT

1.1.1 Transition Test

The idea is to reject the potential nodes that are of high cost. First, configurations whose cost is higher than the maximal threshold cost $c_{\text{max}}$ are filtered. Then, the probability of acceptance of a new configuration is defined by comparing its cost $c_j$ relatively to the cost $c_i$ of its parent configuration. If you find a collision-free node in the low cost space, just take it. But if it is in the high cost space, you have some probability of taking it. One scenario would be you have rejected many nodes in the low cost space, so you want to take some in the high cost space and hopefully that took you nearer to the goal.
Figure 1: Transition-based RRT on a 2D costmap (the elevation corresponds to costs). The exploration favors the expansion in valleys and saddle points connecting low cost regions.

```
input : configuration space CS
    the cost function c
    the start \(q_{\text{start}}\) and goal \(q_{\text{goal}}\)
output: the tree \(T\)
\(T \leftarrow \text{initTree} (q_{\text{init}})\);
while not reaching the goal do
    \(q_{\text{rand}} \leftarrow \) sample configuration in the space;
    \(q_{\text{near}} \leftarrow \) find the nearest neighbor to \(q_{\text{rand}}\) in the tree;
    if Extend \((T, q_{\text{rand}}, q_{\text{near}}, q_{\text{new}})\) then // pass collision checking
        if \((\text{TransitionTest} (c(q_{\text{near}}), c(q_{\text{new}}), d_{\text{near} \rightarrow \text{new}}) \text{ and } \text{MinExpandControl} (T, q_{\text{near}}, q_{\text{rand}}))\)
            then
                addNewNode \((T, q_{\text{new}})\);
                addNewEdge \((T, q_{\text{near}}, q_{\text{new}})\)
        end
    end
end

Algorithm 1: tRRT Algorithm Part I: Overall Flow
```
The parameters involve in the transition tests are slope of the cost, and average cost of the configurations and temperature parameter.

**Slope of the cost:** Downhill transitions are accepted. For uphill transitions, the steeper the slope is (higher cost), the lower chance there is for it to be accepted.

**The average cost of configuration:** The probability is to be normalized by the average costs of start and goal configurations.

**Temperature:** It controls the difficulty level of transition tests. Low temperature eliminates the expansion to high cost regions. When we exhausted the lower cost regions, the temperature increases to allow expansion in higher cost regions.

\[
\text{input} : c_i, c_j, d_{ij}
\]
\[
\text{output}: \text{True or False, change of } T
\]

\[
K \leftarrow \frac{(c_{\text{init}} + c_{\text{goal}})}{2};
A \leftarrow -\frac{(c_j - c_i)}{d_{ij}};
B \leftarrow K \ast T;
\]

if \( c_j > c_{\text{max}} \) then return False;

if \( c_j < c_i \) then return True;

\[
p \leftarrow \exp\left(\frac{A}{B}\right);
\]

if \( \text{rand}(0,1) < p \) then
    \[
    T \leftarrow T/a;
    n_{\text{fail}} \leftarrow 0;
    \text{return True;}
    \]
else
    if \( n_{\text{fail}} > n_{\text{fail,max}} \) then
        \[
        T \leftarrow T \ast a;
        n_{\text{fail}} \leftarrow 0
        \]
    else
        \[
        n_{\text{fail}} \leftarrow n_{\text{fail}} + 1
        \]
end
return False;

**Algorithm 2:** tRRT Algorithm Part II: TransitionTest

1.1.2 Expansion Control

The adaptive temperature tuning introduced above for the transition tests ensures a given rate of slope climbing successes. However it leads to an exhaustion of the lower cost regions before increasing temperature to explore higher cost region. It means a too slow expansion of the tree versus an excessive refinement of explored regions. One idea to solve the problem is to apply expansion control. The details of the method is discussed in [1]

1.2 RRT*

RRT* is another extension of RRT. It guarantees to achieve optimal path. The algorithm expands the tree and optimizes the path at the same time. It optimizes the path by the technic calling **rewiring**: When a new node is added to the tree, we always try to rewire its nearby nodes to achieve lower cost than without the new node. However, it does not have a convergence rate, meaning the planner will give an optimal path if we run it for as long as it needs to. It does not give a path at any time we would like it to, as opposed to Any-time RRT. On the figure 3 is an illustration of the RRT* process.
Figure 2: Impact of the minimal expansion control on the T-RRT algorithm. Without control (left), the insertion of refinement nodes tends to slow down the exploration by decreasing the temperature. By performing this control, the planner is forced to keep exploring new regions of the space.

Figure 3: RRT*-AR: Sampling-Based Alternate Routes Planning with Applications to Autonomous Emergency Landing of a Helicopter, [2]
We start first with RRT procedure: sample one configuration in the space, then find the nearest neighbor to that configuration in the existing tree. For each nearest neighbor $q_i$, we have cost $c(q_i) = c(q_s, q_i) + c(q_i, q_{sample})$, pick $q^* = \arg\min(c(q_i))$, if $c(q_s \rightarrow q_i) > c(q_s \rightarrow q^* \rightarrow q_{sample} \rightarrow q_i)$, rewire the tree $q_s \rightarrow q^* \rightarrow q_{sample} \rightarrow q_i$. The rewiring process happens for every new node added, with no judgement of significance to the goal. It is still an open area of research. The rewiring process is illustrated on the figure 4.

![Figure 4](http://mathoverflow.net/questions/80747/a-rewiring-process-on-graphs)

(a) Starting from this 10-node graph of 20 edges (b) The process first deletes (1,8) and adds (1,10) and (8,5), then continues until the complete graph minus the edge (1,8) is reached

Several improvements to the bare-bone RRT* were proposed:

- Bi-directional RRT*

- Anytime Motion Planning using the RRT

- RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution

From the performances of various extensions of RRT algorithm, we can see the strength of RRT is its rapid exploration. The combination of bidirectional RRT and path shortening usually gives a time-efficient and satisfactory result.

2 Discrete Search

Planning with probability roadmaps is effective with its exploration strength, but can be less suitable for mission-critical tasks and tasks that requires optimal solutions. Another approach to lower the infinite dimension of the world would be to discretize the space with some resolution desired. But, as the dimension increases, the cost of discretization will increase as well. For example, 10 cells in 1 dimensional space will be $10^7$ cells in a 7 dimensional space.
We can try to solve the problem by developing only part of the world that we needed in the planning process. Or we can make the grid very coarse at the start, find a solution in that coarse grid, divide it into finer grid and iterate through to find a desired solution incrementally. A good example of discretized problem would be the puzzle game. The black space needs to go from start (a) to goal (b), through a series of moves. The moves can be [up, down, left, right], and after each move, we have a usually different state. Our goal is to find the path, which is a sequence of moves.

2.1 Traverse of Graphs

We can have different orders to traverse through the nodes in the graph.

- **pre-order**: visit each node before its children.
- **post-order**: visit each node after its children.
- **in-order** (for binary trees only): visit left subtree, node, right subtree.

2.2 Depth First Search (DFS)

Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. DFS will process the vertices first deep and then wide. After processing a vertex it recursively processes all of its descendants.

**Procedure** DFS(starting vertex v)

1. visit(v);
2. foreach neighbor w of v do
   1. if w is unvisited then
      1. DFS(w);
      2. add edge v → w to tree T
   1. end
1. end

**Algorithm 3**: Depth First Search
2.3 Breadth First Search (BFS)

BFS will process the vertices first wide then deep. We just keep a tree (the breadth first search tree), a list of nodes to be added to the tree, and markings (Boolean variables) on the vertices to tell whether they are in the tree or list.

**Procedure** \texttt{BFS()}

- find some vertex \( x \);
- mark \( x \);
- list \( L \leftarrow x \);
- \textbf{while} \( L \) is not empty \textbf{do}
  - choose some vertex \( v \) from front of list;
  - visit \( v \);
  - \textbf{foreach} unmarked neighbour \( w \) \textbf{do}
    - mark \( w \);
    - add it to end of list;
    - add edge \( v \rightarrow w \) to \( T \)
  - \textbf{end}
- \textbf{end}

**Algorithm 4:** Breadth First Search


3 Cost-based Planning

3.1 Dijkstra Algorithm

Dijkstra’s algorithm, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms. Figure 5 shows how this algorithm works.

![Figure 5](http://en.wikipedia.org/wiki/File:Dijkstra_Animation.gif)

Function Dijkstra(Graph, source)
/* at the very beginning, set every node's value (which represent the cost to
reach to this node) to be infinity, and every previous node to be unvisited */
foreach vertex v in Graph do // [h]Initializations
    dist[v] ← infinity; // Unknown cost(distance) function from source to v
    previous[v] ← undefined; // Previous node in optimal path from source
end
/* set the cost (or distance) of the start point to be zero and push all the nodes
into the queue (We could use the data structure min-priority queue, so that
every time we could directly get the node with minimum cost) */
dist[source] ← 0; // Distance from source to source
Q ← the set of all nodes in Graph; // All nodes in the graph are unoptimized thus are
in Q
/* in the main loop, every time pop out the node with minimum cost, check all the
vertex around it, calculate the distance to reach to this vertex. If the
distance is less than the value that node stores, replace the value and record
its predecessor. Repeat the whole procedure until we reach the destination */
while Q is not empty do; // the main loop
    u ← vertex in Q with smallest distance in dist[]; // Source node in first case
    remove u from Q
    if dist[u] = infinity then
        break; // all remaining vertices are inaccessible from source
    end
    foreach neighbor v of u do ; // where v has not yet been removed from Q
        alt ← dist[u] + dist_between(u,v)
        if alt < dist[v] then ; // Relax (u,v,a)
            dist[v] ← alt
            previous[v] ← u
            decrease-key v in Q; // Reorder v in the Queue
        end
    end
end
return dist

Algorithm 5: Dijkstra

3.2 A*

In Dijkstra algorithm cost function \( C(x) \) is a cost to come from initial state \( x_i \) to the current state \( x \). Based
on this function the priority queue \( Q \) is sorted. In A* algorithm new cost function \( G(x) \) is added. This is a
cost to go from current state \( x \) to goal state \( x_g \). Optimal cost to come may be computed, but cost to go may
be only estimated. In order to find optimal path estimated cost to go should always be less than optimal.
To achieve this, heuristic cost is used, which is just a length of straight path from current state to goal state.
Since this path usually contains an obstacle, the actual cost to go will be higher.
Combining these to cost functions we will get:

\[ F(x) = C(x) + \epsilon G(x) \]

The function \( F(x) \) is used to sort priority queue \( Q \). For \( \epsilon = 1 \) this is A* algorithm, for \( \epsilon = 0 \) this is Dijkstra.

An advantage of A* algorithm is that it is guaranteed to perform the least amount of work among other discrete search algorithms. A disadvantage of A* is that it may consume more memory.

4 Bibliography
